Accelerated Cooperative Co-Evolution on Multi-core Architectures

by
Edmore Moyo*

Supervisor : Michelle Kuttel†
Co-supervisor : Geoff Nitschke‡

Dissertation presented in fulfillment of the requirements
for the degree of
MASTER OF SCIENCE
Department of Computer Science,
Science Faculty,
University of Cape Town.

February 20, 2019

*etmoyo@gmail.com
†mkuttel@cs.uct.ac.za
‡gnitschke@cs.uct.ac.za

Abstract

The Cooperative Co-Evolution (CC) model has been used in Evolutionary Computation (EC) to optimize the training of artificial neural networks (ANNs). This architecture has proven to be a useful extension to domains such as Neuro-Evolution (NE), which is the training of ANNs using concepts of natural evolution. However, there is a need for real-time systems and the ability to solve more complex tasks which has prompted a further need to optimize these CC methods. CC methods consist of a number of phases, however the evaluation phase is still the most compute intensive phase, for some complex tasks taking as long as weeks to complete. This study uses NE as a test case study and we design a parallel CC processing framework and implement the optimized serial and parallel versions using the Go programming language. Go is a multi-core programming language with first-class constructs, channels and goroutines, that make it well suited to parallel programming. Our study focuses on Enforced Subpopulations (ESP) for single-agent systems and Multi-Agent ESP for multi-agent systems. We evaluate the parallel versions in the benchmark tasks; double pole balancing and prey-capture, for single and multi-agent systems respectively, in tasks of increasing complexity. We observe a maximum speed-up of 20x for the parallel Multi-Agent ESP implementation over our single core optimized version in the prey-capture task and a maximum speedup of 16x for ESP in the harder version of double pole balancing task. We also observe linear speed-ups for the difficult versions of the tasks for a certain range of cores, indicating that the Go implementations are efficient and that the parallel speed-ups are better for more complex tasks. We find that in complex tasks, the Cooperative Co-Evolution Neuro-Evolution (CCNE) methods are amenable to multi-core acceleration, which provides a basis for the study of even more complex CC methods in a wider range of domains.

Plagiarism Declaration

'I know the meaning of plagiarism and declare that all of the work in the dissertation, save for that which is properly acknowledged, is my own'.

Edmore T. Moyo

Acknowledgements

I would like to express my thanks to my wife Thandazile and my daughters, Angelica and Gabrielle for their support. I would also like to extend my thanks to my co-supervisor Geoff Nitschke for introducing me to the world of Neuro-Evolution. To my supervisor Michelle Kuttel, I would like to extend my deepest thanks for your wisdom and perseverance throughout the entire process. Your commentary and advice has greatly improved my research skills.

Finally I would like to thank the CILT writing circle for their feedback and the University of Cape Town's ICTS High Performance Computing team: `http://hpc.uct.ac.za`, whose facilities I used to perform my computations.

Contents

List of Tables

List of Figures

1 Introduction

An artificial neutral network (ANN) [27] is a massively parallel distributed processor made up of simple processing units that resembles the human brain. Like the human brain, knowledge is acquired through learning and the acquired knowledge is stored. Supervised learning, where the desired output is already known, is commonly used to train ANNs. However, in the last two decades an unsupervised method that *evolves* ANNs, Neuro-Evolution (NE) [41], has grown in popularity due to its efficiency in solving real world tasks [22]. NE uses Evolutionary Algorithms (EAs) [17] to evolve ANNs, in order for them to perform a task. However for complex tasks this training process is time consuming and a speed-up of even an order of magnitude will enable simulations to be run at a faster rate which could prove cost effective and be suitable for real-time systems. A real-time system may be defined as one in which the timing requirements are an essential part of its specification. One common example is industrial process control such as a chemical plant or active rocket guidance systems [23] which tend to have real-time requirements. This speed-up can potentially be achieved by harnessing parallel computing [24].

In the past most software developers relied on advances in hardware to automatically increase the speed of their applications, however, this is no longer the case. Around 2003, there was a shift towards multi-core processors in order to reduce power consumption and heat dissipation issues in the processor architectures at the time. These issues caused a limitation in the productive processes that can be performed within each clock period within a single Central Processing Unit (CPU) [56]. In the future in order to truly leverage the advances in hardware, software developers need to write programs that utilise the available processor cores.

As described by Kirk and Hwu [37], designing the multi-processor has taken two trajectories: the multi-core path and the many-core path. The multi-core trajectory has focused on maintaining the execution speed while transitioning into multiple cores, for example multi-core processors whilst the many-core path has focused on execution throughput of parallel applications, for example the Graphics Processing Units (GPUs). The CPU is optimized for sequential code performance whereas the GPU is designed to be a numeric computing engine, therefore an application that includes data parallelism is well suited to GPUs. Data parallelism refers to a program property whereby many arithmetic operations can be safely performed on a data structure in a simultaneous manner, an example is matrix multiplication.

The success in speeding up parallel tasks using hardware has led to the development of libraries and languages that leverage either multi-core or many-core architectures, or both, such as Chapel [5], Threading Building Blocks (TBB) [29], MPI [47], CUDA [37] and Go [38]. Even with the growing number of libraries available, selecting the right tool for a complex task still remains a challenge. Developers still find themselves porting parallel programs from one library or language to another in order to run on different parallel architectures. Such porting of code has a cost implication as it usually entails rewriting large parts of the code and it is also error prone. In some instances, tasks may have similar algorithms but if the implementations are not sufficiently general then reusing the already written codebase may be a difficult task. The

Go programming language has emerged in recent years and makes coding for the multi-core easier, in that it has high level abstractions, that alleviate the programmer from concerning themselves with threaded programming [45]. These abstractions enable tasks to be run in parallel and to be easily synchronized. This is a very applicable pattern to the implementation of parallel algorithms. Portability is also a key issue affecting parallel computing, Go addresses this issue through advanced dependency management (compiling of programs into a single executable binary) and cross compilation (ability to compile a program to be run on multiple architectures), which enables a Go program to be run on multiple parallel architectures.

If useful environmental parameters are gathered at a fast rate, such fast learning could greatly benefit real-time systems. Within the NE community, the focus has been on improving algorithms or the creation of novel methods to evolve ANNs [22, 64] to efficiently search the state space of a problem (a set of states that a problem can be in). NE methods have been successful because of their ability to search large state spaces and to solve non-linear and partially observable tasks (for example, tasks that have incomplete state information). The time to search large state spaces is still an overhead, as it can take up to several days to solve very complex tasks. However, an evolutionary approach, Cooperative Co-Evolution (CC) [48] enables a problem to be broken down into manageable sub-components and has been applied successfully to the domain of NE to decompose large state spaces [23], so that they are solved concurrently. For this reason,CC has provided a useful extension to NE. NE is a powerful approach to solving standard benchmark tasks, such as pole balancing and prey-capture [59, 43], but also real-world tasks, such as rocket guidance, robot control, game playing, resource optimization, music generation and modeling language evolution [49, 22, 6, 8, 25, 23, 21, 15]. Another issue affecting the training of ANNs is cost. When an ANN is trained to perform a control task such as balancing a pole or guiding a rocket, the task execution is initially simulated and the parameters necessary to perform the task are then transferred to a physical model. This limits the need for multiple trials on physical models, which can prove to be costly and also reduces human interaction for dangerous tasks. Actually very few ANNs are then tested in counter-part physical models, as they are easily transferrable [23]. If an ANN can be trained to perform a task at a faster rate this could have a great cost benefit, in that less time is spent in the simulation phase.

Cooperative Co-Evolution NE (CCNE) methods are population based, that is, each population is comprised of individuals. Unlike Conventional Neuro-Evolution (CNE) the individuals form partial solutions, that is the individual is not a complete ANN. Individuals are genotypes which are the encoding of the ANNs. Research has shown [51] that decomposing the search space into sub-genotype spaces, ensures that CCNE methods perform more efficient search than CNE methods for large and complex search spaces. Most CCNE methods follow this process: 1) a population of individuals is generated, 2) individuals are then selected to form an ANN or a team of ANNs and, 3) the ANN (or team of ANNs) is then evaluated within a task environment.

In general, each individual participates in a certain number of network evaluations to gauge their average fitness within the task. Their fitness is then used to evolve the population of individuals, that is, the individuals with the greater fitness scores mate to form new stronger individuals

to replace the weaker ones. The evolutionary process continues until an ANN optimized for the particular task is found. The network evaluations, because they are a series of trials, can be CPU intensive for very complex tasks, that require a large number network evaluations to complete. However, because evaluations can be carried out independently they lend themselves naturally to parallelization. One promising approach is the combination of CC and hardware, to spread evaluations across multiple cores or processors, so that they are carried out in parallel [22].

1.1 Motivation

A review of the literature [23, 1, 10, 57, 46, 4, 40, 34] indicated that there is potential to exploit parallelism to improve the performance of CC methods. Much of the focus has been on making adjustments to synchronous CC algorithms to make them asynchronous or on performance gains for specific CC algorithms [46]. Although these implementations have yielded some performance gains, there are some issues to be addressed: 1) there are limitations to how much an algorithm can be fine-tuned to make it asynchronous, particularly if it does not leverage all the available cores, 2) if the asynchronous algorithms are specific to a particular domain then they are not easily transferrable to other CC methods, 3) if the implementations are specific to a particular hardware then they are not easily portable, and 4) it still takes a long time to evolve CC methods in complex tasks. Complex tasks are highly dimensional, with a large search space and therefore are compute intensive.

Recent surveys [40, 34] have also highlighted other gaps in the state-of-the-art, such as lack of optimal design of algorithms, scalability issues, lack of re-usability of frameworks, lack of theoretical foundations in CC research and lack of agreed performance criteria. These are some of the gaps in the state-of-the-art that this thesis would like to address.

In the literature reviewed one interesting observation was that the implementations were multi-threaded and ran on specific hardware [1, 34]. Portability is a key issue that affects parallel computing and in this study we will implement optimized serial and parallel versions in the Go programming language, as it has high level abstractions that alleviate the programmer from multi-threaded programming, and can be run on different parallel hardware architectures.

1.2 Aims and Objectives

The overall aim of this study is to provide a foundation for the parallelization of CC methods on multi-core architectures. We aim to identify a parallelization approach for accelerating CC methods that is sufficiently fast, portable and general. *Fast* refers to the parallel speed-up of the evolution time of CC methods versus single processor evolution time. *Portable* refers to the ability for the implemented algorithm to be run on different parallel architectures. Finally *general* refers to the implementation of multiple NE methods in a parallel processing architecture as demonstrated by three case studies: Symbiotic Adaptive Neuro-Evolution (SANE) [44], Enforced SubPopulations (ESP) [22] and Multi-Agent ESP [64]. The objective of this work is to

empirically compare serial versions of three NE methods, namely SANE, ESP and Multi-agent ESP to their parallel versions in the benchmark tasks: double pole balancing and prey-capture.

1.3 Approach

Several CC methods exist but the prime focus of this dissertation is the Neuro-Evolution (NE) domain. NE is selected as a test case study given its wide use and broad problem solving capabilities [41]. The approach is phased as follows: 1) first we design a parallel processing framework, 2) we implement the optimized serial versions of three NE methods: SANE, ESP and Multi-Agent ESP, 3) we implement the parallel versions of the NE methods, by applying the parallel processing framework to the serial versions, 4) we validate the serial and parallel versions and, 5) we benchmark the parallel versions in the double pole balancing (for SANE and ESP) and prey-capture (for Multi-Agent ESP) tasks, to demonstrate the performance gains when the parallel processing framework is applied. A single population version of SANE, known as neuron SANE is implemented as part of our work, in order to evaluate the performance advantage of the CC architecture over NE-only approaches.

1.4 Contribution

The contribution of this work is three-fold: 1) we propose a parallel processing framework for CC methods, that takes advantage of the concurrent nature of CC, combines that with the MapReduce model [13] and leverages the power of parallel computers, 2) an empirical study that compares the different versions of the NE methods in benchmark tasks: double pole balancing and prey-capture, and 3) optimized serial and parallel implementations of the NE methods: SANE, ESP and Multi-Agent ESP.

1.5 Thesis Overview

Chapter 2 provides the background necessary to understand the current state of parallel computing and how it relates to increasing the performance of CC methods. First we introduce parallel computing, the current issues affecting it and how parallel programming languages, such as Go address some of these issues. We then give an overview of the CC model, the building blocks of NE and how NE methods are evaluated. The next section then gives an overview of CCNE methods, and details three case studies: SANE, ESP and Multi-Agent ESP. We then conclude the chapter with a literature review of the current state-of-the-art with regard to the parallelization of CC methods.

Chapter 3 contains a detailed description of the design of the CC parallel processing framework, its application to the case study NE methods, and the design and implementation challenges encountered.

Chapter 4 reports on the validation of the NE case study methods in the validation tasks: double pole balancing (for SANE and ESP) and prey-capture (for Multi-Agent ESP).

Chapter 5 then presents the results of the benchmarking of the case study methods in the benchmark tasks: double pole balancing and prey-capture, on a High Performance Computing

(HPC) cluster.

Chapter 6 discusses the contributions of this study and suggests future work.

Appendix A contains the hardware specifications for the validation and benchmarking experiments. **Appendix B** contains snapshots of our visualizations of the double pole balancing and prey-capture tasks. **Appendix C** contains the actual runtimes for the validation experiments performed on the HPC cluster. **Appendix D** contains the actual runtimes for the benchmarking experiments performed on the HPC cluster and **Appendix E** lists the information required to access the Go implementations of the CCNE methods.

2 Foundations

In this chapter we provide some background on parallel computing, Neuro-Evolution (NE) and Cooperative Co-Evolution (CC). We also give an overview on the current state-of-the-art and how our work will contribute to enhancing it.

2.1 Parallel Computing

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. The compute resources are normally a single computer with multiple processors or cores, or a number of such computers connected by a network, for example a High Performance Computing (HPC) cluster. The computational problem is partitioned into manageable parts that are then then solved simultaneously. Parallel compute resources have been present for a long time in the form of HPC, which was normally reserved for the scientific community, due to the high cost implications, however they are now more prevalent in modern computer architecture. To run parallel programs on a parallel computer, they need to be implemented in a programming language, this language requires extra mechanism for it to be able to share information among the processors or processor cores [24]. To enable this a variety of programming paradigms have been developed. There are several parallel programming paradigms and different parallel programming languages enforce different programming paradigms. According to Grama [24], three main factors influence the variations: 1) effort invested in writing the parallel programs, 2) efficiency on certain parallel computer architectures, and 3) various applications have different types of parallelism, so different languages exploit them. Examples of parallel programming paradigms are: explicit, implicit, shared-address-space, message-passing, data parallelism and control parallelism.

Explicit parallel programming is an approach where a parallel program is explicitly written. This means that the programmer has to specify how the processors or processor cores will cooperate in order to solve the specified problem. This requires extra effort as the programmer needs to determine the communication and synchronization points and explicitly program them. Another approach is called implicit parallel programming, where the programmer uses a sequential programming language but it is up to the compiler to insert the parallel constructs to run the program on a parallel computer. This type of automatic parallelization though difficult in the past [24], is now possible through the use of parallel programming models such as OpenAcc [60].

Another paradigm is shared-address-space, this is where data is stored in a globally accessible memory and each processor accesses the shared data by reading or writing to shared variables. The sharing of variables makes this paradigm subject to mutual-exclusion problems and therefore the programming languages that employ this paradigm need the appropriate constructs to mitigate this. In contrast to the shared-address-space paradigm, in the message-passing paradigm the programming languages have an ability to send and receive data among processors and cores and there is an emphasis on private and local variables [24].
Data parallelism is when the same operation is performed on a data set by assigning data

elements to various processors or processor cores. One common example of a data-parallel problem is matrix multiplication. Task parallelism (or control parallelism) on the other hand focuses on distributing tasks across multiple processors or cores. Many problems exhibit a certain amount of both data and task parallelism, however problems that do not exhibit data parallelism are better suited to multi-core architectures whereas data parallelism is better suited to Graphics Processing Units (GPUs).

2.2 Issues in Parallel Computing

In order to fully utilize parallel computing a number of issues need to be addressed. One issue is the design of parallel computers. Currently most vendor computers on the market are multi-core computers and this is the area in parallel computing that has seen the most advances. The most important aspect in their design remains their ability to scale up to a large number of processors and support communication between those processors or processor cores [24]. Another issue is the ability to design efficient parallel algorithms. This entails the ability to effectively break down a serial program into concurrent parts that can be executed in parallel. In order to effectively gauge the performance gains due to parallelization, it is necessary to be able to evaluate them; this is one aspect that is also important in parallel computing. Evaluation allows us to determine how fast problems are solved and also how effectively the processors or processor cores are utilized. Also of importance is parallel programming languages and tools. Parallel programming languages are used to implement parallel algorithms, and tools, such as debuggers, profilers and simulators are used to ensure correct and effective programming. The language must provide both programming efficiency and tooling, one such language is Go [38].

Go is a general purpose programming language that was created to combine efficient compilation with efficient execution and ease of programming which its creators felt was lacking in already existent languages. Go's concurrency model is inspired by Communicating sequential processes (CSP) [31], unlike other languages that support concurrency as an extension or library, Go has built in concurrency. CSP is an algebraic formalisation for describing how concurrent systems interact. Its support for concurrency is based on two first class constructs, *goroutines* and *channels*. The two constructs are what we will leverage to implement an efficient and general parallel processing framework. A goroutine is a lightweight thread managed by the Go runtime. The *go* statement starts the execution of a goroutine, for example, to start a new goroutine running *f(x,y,z)* :

go f(x,y,z)

Goroutines have several advantages over conventional threads, for example you can run more goroutines on a typical system than you can conventional threads, because they are light-weight. Goroutines can safely communicate with one another using another built in construct, known as a channel. Channels are a typed conduit through which you can send and receive messages. Channels allow for *synchronization* without the need for explicit locks or condition variables, which alleviates the programmer from low level thread programming. Channels can be buffered

or unbuffered. This approach ensures that only one goroutine has access to the data at a given time [38].

Parallelization in Go is achieved via the Go runtime and a setting; *GOMAXPROCS*. This setting allows a programmer to determine the number of logical CPUs (the number of physical cores x the number of threads, that can run on each core through the use of hyper-threading) to be used by a program. The Go runtime schedules goroutines to run within a logical processor that is bound to a single operating system thread [38]. The GOMAXPROCS setting is by default set to one, which allocates a single logical processor to execute all goroutines in a program. In order to achieve true parallelism the GOMAXPROCS value must be increased and the scheduler has to distribute goroutines across logical CPUs on a machine with multiple physical processors. Another issue that Go addresses is portability, as it can be cross compiled to run on different operating systems and parallel computers.

There are a variety of overheads associated with parallelism [18, 50]. These overheads result in the ideal speed-up (a speed-up of N on N cores) not being achieved. A parallel program may have to perform extra activities besides those performed by its serial counterpart. This results in overheads which may be a result of 1) excess computation, activities not performed by the serial program, 2) interprocess communication, 3) time spent idling, 4) inefficient scheduling of work, and 5) a program not having enough parallelism. Interprocess communication refers to the communication and exchange of data by processing elements in parallel programs, this accounts for most of the processing overheads. Idling is when processing elements have to wait for other processing elements to complete their work before they resume, this may be caused by synchronization points or uneven distribution of work. Excess computation is another source of overhead, because a parallel program may have to carry out extra computation, for example, if some computation that is normally carried out serially has to be repeated on the processing elements. Runtime schedulers are responsible for creation and scheduling of work, however if the work that is partitioned is too small, then the cost of scheduling the work may be too large. At times however, a serial program may not have enough parallelism or the task it is meant to solve may be too simple, such that the cost of parallelism is large.

In recent work Nanz et al. [45] benchmarked the usability and performance of four multi-core programming languages in six benchmark programs. Of particular interest to this study is the analysis that was done on Go. Nanz et al. reported a speed-up deterioration for Go in two benchmarks tasks. They attributed this deterioration to excessive creation of goroutines, generating scheduling and communication overheads. This analysis indicates that a Go program may experience a deterioration in speed-up if the Go runtime has to schedule multiple goroutines across multiple cores, that need to synchronize and exchange data.

2.3 Cooperative Co-Evolution

Co-Evolution in natural ecosystems refers to how organisms in a species either cooperate or compete with other species in order to survive. This concept has been adopted in the domain

of Evolutionary Computation (EC) to solve complex problems and there are two main types of Co-Evolution approaches, competitive and cooperative.

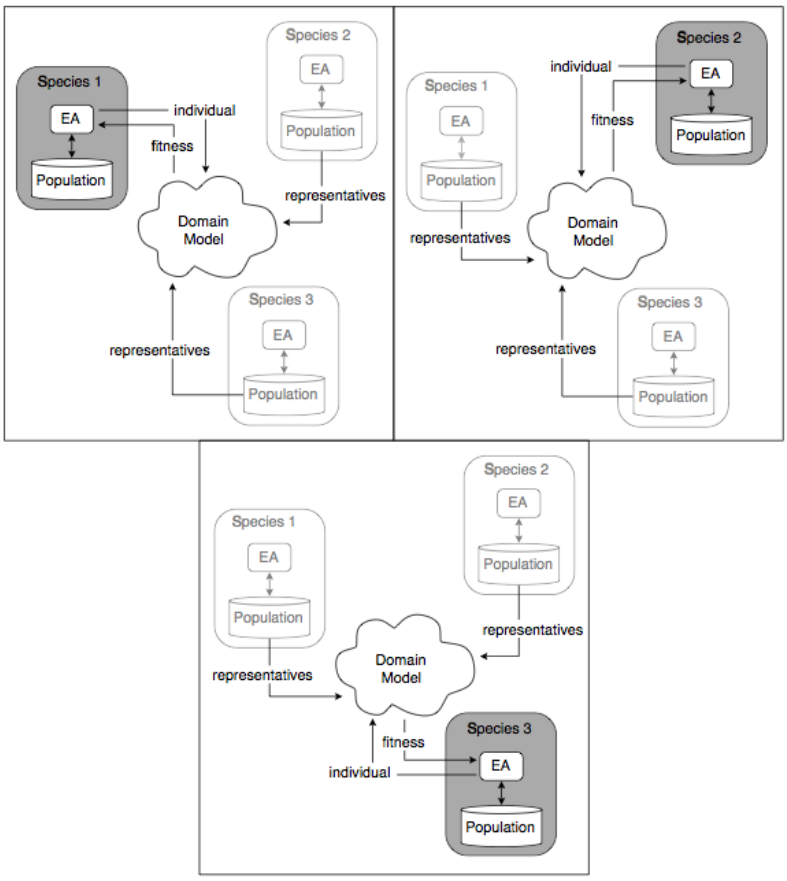


Figure 2.1: **The Cooperative Co-Evolution Framework**. Three species are co-evolved in a problem domain. Figure taken from Potter and Jong [48].

Competitive Co-Evolution [55] in Evolutionary Computation (EC) focuses on competition, where species go head to head in a sort of arms race producing increasingly stronger strategies for the others to defeat. Cooperative Co-Evolution [48] on the other hand emphasizes cooperation, and will be the focus of this study.

Cooperative Co-Evolution may be defined as a generalized architecture (Figure 2.1) for evolving interacting co-adapted sub-components. The ecosystem consists of two or more species that, as in nature, only mate with members of their own species. Mating restrictions are enforced by evolving the species in separate populations [48]. Each individual within the species represents a partial solution and the complete solution is formulated through cooperation within the tasks. This modularization means that tasks can be broken down into sub-problems and optimized concurrently. After evaluation only the most beneficial in solving the task are recombined and the weaker individuals are discarded. This architecture is also heterogeneous in nature, meaning heterogeneous representations are supported, that is, components may be implemented as artificial neural networks (ANNs), whilst others may be implemented as symbolic rules. Therefore depending on the problem the most appropriate evolutionary process and representation

can be selected. These characteristics have enabled this architecture to be applied to different domains, one such domain is Neuro-Evolution (NE), which we will describe in the next section.

2.4 Neuro-Evolution (NE)

Neuro-Evolution (NE) [41] is a form of machine learning where ANNs are modified using Evolutionary Algorithms (EAs) [17] in order to learn a specific task. NE has been shown to be robust and efficient at solving complex real world tasks, in areas such as robotics [15] and control [23].

One important feature of NE is its ability to adapt a broad range of ANN types, as opposed to other ANN training methods. It is also able to modify weights, topologies (architectures) and even teams of ANNs, which makes it a more general, and hence broadly applicable method. The following subsections introduce the building blocks for Neuro-Evolution (NE). NE is an unsupervised approach whereas previous approaches have been supervised.

2.4.1 Artificial Neural Networks (ANNs)

ANNs are massively parallel processors inspired by biological nervous systems, in particular the animal brain. They model the way in which the brain performs a particular task or function of interest [27]. ANNs have the ability to learn as well as to store the acquired knowledge.

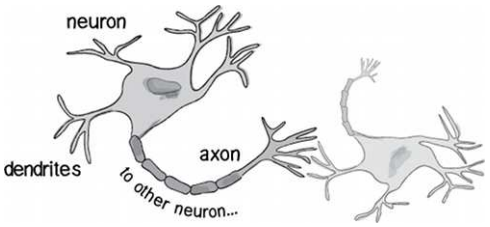


Figure 2.2: **A biological neural network**. Figure taken from Shiffman, The Nature of Code [52].

Processing units known as neurons form the basis of ANNs, like biological neurons (Figure 2.2), they are also inter-connected. The neurons form a layered structure and the number of layers, input neurons or output neurons may be varied depending on the complexity of the task. For example, for a three layer neural network such as the one shown in Figure 2.3a, the first layer has input neurons which receive information from the environment that then activates the input layer. The input layer then sends data via weighted synapses to the second layer of neurons, then via more weighted synapses to the third layer of output neurons.

Each neuron i performs a transfer function f_i of the form:

$$y_i = f_i(\sum_{j=1}^{n} w_i x_j - \theta_i)$$

where y_i is the output of the neuron i, x_j is the jth input to the neuron and $w_i j$ is the connection

weight between the neurons i and j. θ_i is the threshold (or bias) of the neuron. Usually f_i is non-linear, such as Heaviside, Sigmoid, or Gaussian function [63].

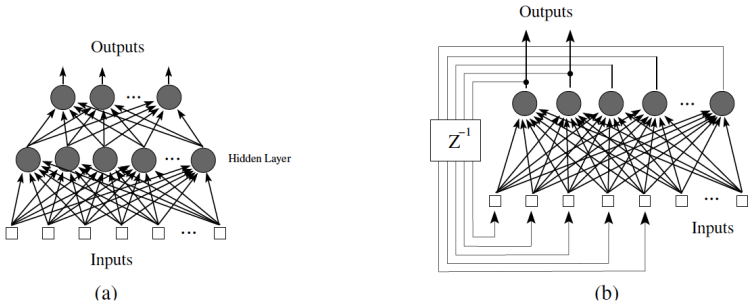


Figure 2.3: **Types of artificial neural networks**. a) A feedforward network, no feedback connections. b) A recurrent network, with feedback connections and hence a short-term memory. Figure taken from Gomez and Miikkulainen [22].

The network described above is known as a feedforward network, as information is projected or fed forward and never vice-versa. Another type of network with cycles or feedback connections is known as a recurrent network (Figure 2.3b). Recurrent networks can retain information, and are well suited for tasks that require short-term memory, for example tasks that are partially observable (not all state information is available). ANNs were traditionally trained using gradient-based methods such as back-propagation [30], but more recently EAs have grown in popularity due to their efficiency and broad capabilities.

2.4.2 Evolutionary Algorithms (EAs)

Evolutionary algorithms (EA) [17] are a family of stochastic problem solvers based on principles that can be found in natural evolution [16]. EAs are a superclass encompassing Genetic Algorithms (GAs) [32], Evolution Strategies (ES) [26] and Evolutionary Programming (EP) [19].

All of these algorithms use a population based search strategy. A general algorithm for EAs is shown below [63]:

1. Generate the initial population *G(0)*, and set $i = 0$;
2. REPEAT
   (a) Evaluate each individual in the population;
   (b) Select parents from *G(i)* based on their fitness in *G(i)*;
   (c) Apply search operators to parents and produce offspring which form *G(i+1)*;
   (d) $i = i + 1$;
3. UNTIL 'termination criterion' is satisfied.

Evolving ANNs using EAs has been shown to provide solutions to complex real world problems, as they are less likely to be trapped in local minima than traditional gradient-based search algorithms [63]. The next section details the application of EAs in the evolution of ANNs.

2.4.3 Using EAs to evolve ANNs

It has been shown that combinations of ANNs and EAs can lead to significantly improved task performance than relying on ANNs or EAs alone [63, 14]. Evolution of ANNs can be introduced at various levels, namely: 1) Connection weights, 2) Architectures and, 3) Learning Rules. This study focuses on the evolution of connection weights. Connection weights can be represented as binary strings or real numbers and search operators such as crossover and mutation can be used [63]. Two encoding schemes exist: 1) direct and 2) indirect. For direct encoding the details are explicitly encoded on the genotype (or chromosome), whereas in indirect encoding only the important or implicit details are encoded on the genotype (or chromosome).

Many methods evolve connection weights only [23], but some evolve architectures (topology) as well [44, 55]. Evolving neural networks using EAs is computationally intensive, as EAs are global search procedures, that is, they evaluate a larger part of the search space. However approaches such as CC provide an extension to EAs, enabling complex problems (with large state spaces) to be decomposed into manageable parts that can potentially leverage the increasing power of parallel processing computer architectures.

2.5 Evaluation of NE methods

The following sections introduce two benchmark tasks commonly used in the evolutionary computation community to evaluate NE methods. These tasks maintained longevity due to their intuitive appeal and the ability to add variations to them to increase or decrease complexity.

2.5.1 The Pole Balancing Task

The pole balancing problem has been the *de-facto* standard for benchmarking algorithms, in areas such as control theory [58], for many years. Although it is referred to using different names in texts, for example: broom balancer, inverted pendulum, cart-pole, it is still inherently the same task. It describes a general class of unstable systems and it is simple to visualise.

A basic pole system consists of a pole hinged on a wheeled cart on a finite track (Figure 2.4). The objective is to apply a force, either left or right, to the cart at regular intervals such that the pole does not fall over and the cart remains within the track. The state of the system is described by the variables below:

$$x : \text{position of the cart}$$
$$\dot{x} : \text{velocity of the cart}$$
$$\theta_i : \text{angle of the } i\text{-th pole } (i = 1,2)$$
$$\dot{\theta}_i : \text{angular velocity of the } i\text{-th pole}$$

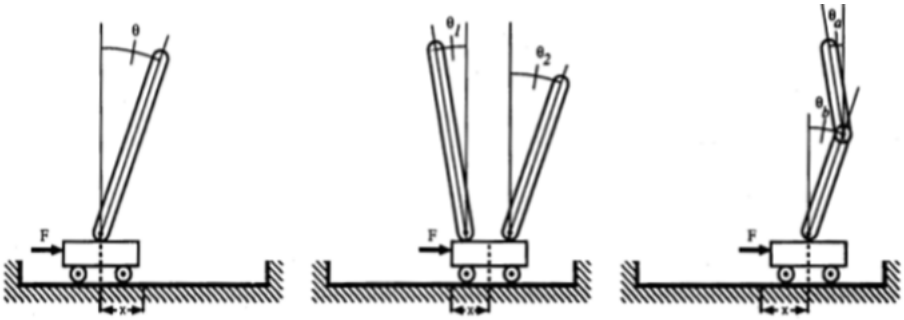


Figure 2.4: **The pole balancing task**. The first problem, on the far left, shows a single pole balanced on a cart. The second problem, in the middle, shows two poles balanced on a cart. The third problem, on the far right, shows a jointed pole balanced on a cart. F represents the force applied to the cart. θ_1 and θ_2 represent the angles of the poles with respect to the cart. Figure taken from Wieland [59].

In early work the pole balancing problem was used to demonstrate the capability of a single artificial neuron, ADALINE (Adaptive Linear Neuron) in controlling an unstable system [58]. A motorized cart carrying an inverted pendulum was assembled and controlled by ADALINE. The single neuron controller, which was termed a broom balancer, was required to keep the single pendulum balanced and to keep the cart within certain bounds by applying a horizontal force to the cart. The training of the ADALINE was supervised, the controller learnt by observing a *teacher*.

Unsupervised approaches such as NE are now able to evolve ANNs to solve the pole balancing problem, without the need of a *teacher*. For example Gomez [21] used the pole balancing task to evaluate how effectively Enforced Subpopulations (ESP) evolved ANNs. This allowed ESP to be evaluated at different levels of complexity, which is a good attribute of the pole balancing problem. Gomez asserts that the reasons for its longevity lay with its intuitive appeal, that is, the fact that it is easy to understand and to visualise, and the fact that it embodies aspects of a whole class of learning tasks that involve temporal credit assignment [21]. However the basic (single) pole balancing task is too simple a task as it can be solved by a single neuron and some NE methods find a solution within even the initial random population [21]. Wieland [59] suggested some variations to make the task harder. To make it more challenging, one extension is modification of the system. This may involve adding an extra pole to make the task non-linear or restricting state information, for example not providing the velocity of the cart or the angular velocity of the pole. This requires that the network learns to compute the difference between the previous inputs and the current inputs, in order to solve the task. To do this memory capability is required, this means that ANNs with this ability are then required to solve these tasks, such as recurrent networks. It has been determined that the double pole balancing task (with incomplete state information) becomes more difficult as the poles assume similar lengths [59]. This is termed the non-Markov task.

The most often used task parameters were the ones described by Barto et al. in 1983 [3]. However, some studies [20] have made efforts to standardise the problem as they believed that

track limits	$\pm 2.4m$
failure angles	$\pm 12^\circ$
gravity (g)	$-9.8m/s^2$
length of pole ($2l$)	$1m$
mass of cart (m_c)	$1.0kg$
mass of pole (m_p)	$0.1kg$
magnitude of force (F)	$10.0N$
integration time step	$0.02s$

Table 2.1: **Common pole balancing parameters**. There are however many variations to these parameters.

published results were difficult to compare and that for most of the methods there is no clear evidence of better performance than a random search method. Geva and Sitte [20] suggested that often used parameters (Table 2.1) be kept, except friction (in the equations of motion) and the failure angle of 12°. They stated that in a real cart-pole, friction was difficult to determine and that the failure angle was too restrictive. A failure angle of 90° was recommended. They also suggested initial state variables for the task.

2.5.2 The Prey-capture Task

The prey-capture task is a special kind of pursuit-evasion problem [43] that has been used widely to test multi-agent behaviour. It, like the pole balancing task is easy to reason about as it is similar to what can be found in nature. These tasks are complex in that they require coordination with respect to the problem domain and although found everywhere in natural ecosystems, they are still a challenge for even to the most effective learning systems. Such tasks normally consist of one or more predators moving around a simulated environment attempting to capture prey. These tasks are made considerably harder if the predator has a tactical disadvantage, for example if the predator has no knowledge of the position of the prey, but rather has to guess or has to remember the prey's last position. Yong and Miikkulainen [64] used a variant of the prey-capture task to evaluate the multi-agent method, Multi-Agent ESP. In other variants of the task, the predators are required to surround the prey in order to capture it [28] and the prey moves either randomly or slower than the predators. In Yong's variant it is enough for one predator to capture the prey by occupying the same grid cell as the prey and the prey moves at the same speed as the predators and always away from the closest predator. This means that the only way the predators can catch the prey has to be coordinated and cooperative. In this study, we base our experimental setup on Yong's variant.

2.6 Applications of Cooperative Co-Evolution in Neuro-Evolution

In early work, Holland and Reitman [33] explored extensions to the basic evolutionary model to support co-adapted sub-components in their classifier system. In this system a population of rules was evolved to provide a complete solution based on how well the rules interacted. The

disadvantage of this work was that it was not general, that is, applicable to a wider range of representations and domains. Therefore in later work Potter and Jong [48] developed a general architecture, Cooperative Co-Evolution. Another approach, Neuro-Evolution (NE) has also prompted the extension of the basic evolutionary model, one example being Conventional Neuro-Evolution (CNE). This is the standard NE approach that usually consists of a single population of chromosomes (for example complete ANNs), evolved in a task [59, 23]. In challenging tasks the CNE approach suffers from premature convergence; where the evolution of the population converges at a non-optimal solution. To alleviate these issues a more powerful idea is the combination of CC and NE to solve more complex tasks through improved search efficiency. Instead of running the evolution at a complete solution level like CNE, the evolution is run at the level of partial solutions. The next section introduces three such CCNE methods: Symbiotic Adaptive Neuro-Evolution (SANE) and Enforced SubPopulations (ESP) and Multi-Agent ESP.

2.6.1 SANE

Unlike CNE methods, Symbiotic Adaptive Neuro-Evolution (SANE) [44] is a cooperative co-evolutionary method that instead of evolving a population of ANNs, evolves two populations simultaneously: a population of neurons and a population of network blueprints that specify how the neurons are combined to form complete ANNs. SANE uses *symbiosis*, the interaction of the individuals in the population, to find complete solutions to a task. Since individuals only form a partial solution, this ensures that the population remains diverse, as those individuals assume a kind of specialization. This diversity prevents convergence in the population. SANE can therefore find solutions faster and tackle harder problems than conventional NE methods. Moriarty and Miikkulainen [44] carried out some experiments to compare SANE's symbiotic evolution to more standard neuroevolution techniques. To achieve this four evolutionary approaches were tested in the Khepera simulator (1) SANE, (2) a standard neuroevolution approach using the same aggressive selection strategy as SANE, (3) a standard neuroevolution approach using a less aggressive, tournament selection strategy, and (4) a version of SANE without the network blueprint population. The fourth evolutionary approach, that he termed, *neuron* SANE, is a symbiotic neuron search without the higher level blueprint evolution. Instead of using a population of network blueprints to form the neural networks, neuron SANE forms networks by randomly selecting subpopulations of neurons. This was done to effectively gauge the contribution of the blueprint-level evolution to the SANE method. In this study we take a similar approach and implement neuron SANE to gauge the contribution of CC to NE methods. A big drawback of SANE is its inability to evolve recurrent networks. This is a serious drawback as most interesting tasks require memory [23]. Figure 2.6 shows the evolutionary process of SANE in a task environment. Our neuron SANE algorithm proceeds as follows:

1. Initialization: Select neurons randomly from the population.
2. Create an artificial neural network (ANN) from the selected neurons.
3. Evaluation: Evaluate the ANN in the given task.
4. Add the network's score to each selected neuron's fitness variable.
5. Repeat steps 2-5 a sufficient number of times, for example 10 trials per neuron.

6. Get each neuron's average fitness score by dividing its cumulative fitness values by the number of ANNs in which it was evaluated.
7. Recombination of the population. Perform crossover and mutation operations on the population based on the average fitness value of each neuron. Discard the weaker neurons.

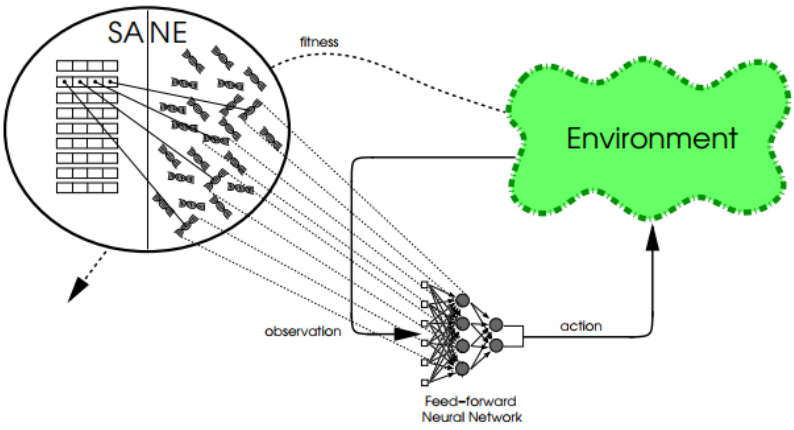


Figure 2.5: **The SANE method**. A population of neurons and network blueprints is evolved in a task environment. Neurons are selected randomly to form a feedforward network that is evaluated in a task environment. Figure taken from Gomez and Miikkulainen [22].

Gomez and Miikkulainen [23] extended SANE and presented a novel CCNE method, ESP, which fully supports recurrent networks and solves tasks that make use of short term memory. The next section describes ESP.

2.6.2 ESP

The ESP method, like SANE is a neuron level CC method, that is, the individuals evolved are neurons and not complete networks. ESP however has the individuals grouped in *subpopulations* and an individual can only be recombined with individuals in its population (Figure 2.6). An individual in ESP is a neuron chromosome consisting of real numbers that represent the connection weights. This specialization, grouping of neurons in subpopulations, allows ESP to search more efficiently than SANE and also to evolve recurrent networks. Gomez and Miikkulainen [23] showed how effectively ESP could be used to evolve an active rocket guidance controller [23], that could be easily transferred to the real world. Transfer means that the controller trained in the simulation environment can be easily used in a real world scenario, with minimal effort.

He also showed that ESP was able to find a solution to a difficult task, the double pole balancing task, relatively faster than Conventional Neuro-Evolution (CNE) methods and other CC methods such as SANE. According to his results ESP was roughly similar to Neuro-Evolution of Augmenting Topologies (NEAT) [55], a direct method that evolves connection weights and topologies (architectures), in terms of performance. NEAT [53] unlike ESP is a Competitive

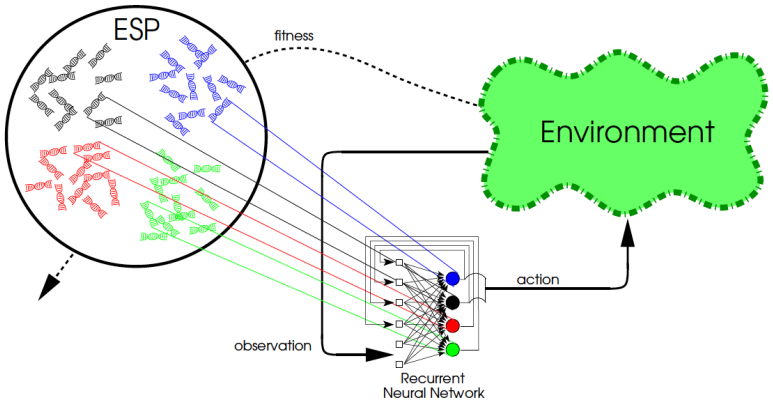


Figure 2.6: **The ESP method**. Subpopulations of neurons are co-operatively co-evolved in a task environment. Due to its broad capabilities ESP can also evolve recurrent networks. Figure taken from Gomez and Miikkulainen [22].

Co-Evolution method. ESP is however simpler, in that it requires far fewer user parameters for each run. The ESP algorithm proceeds as follows:

1. Initialization: h subpopulations are created and initialized with n individuals (neurons).
2. Evaluation: h neurons are selected at random, one from each subpopulation, to form the hidden units of an artificial neural network (ANN). The ANN is evaluated in the task and awarded a fitness score. The score is added to the cumulative fitness of each neuron that participated in the ANN.
3. Check Stagnation: When the ESP method stops making progress towards a solution after b generations, burst mutation is performed. Burst mutation is a means of improving the current population using the current best solution, by adding noise to the current population. If the fitness has not improved after two burst mutations the network size is adapted.
4. Evolution of the population of neurons (recombination): The average fitness of each neuron is calculated by dividing its cumulative fitness by the number of trials in which it participated. Neurons are then ranked by average fitness within each subpopulation. Each neuron in the top quartile is recombined with a higher ranking neuron using 1-point crossover and mutation at low levels and the weaker performing neurons (the lower-ranking half) are replaced by the stronger ones.
5. Repeat steps 2-4 until an ANN that performs sufficiently well in the task is found. A neuron should participate in a trial a sufficient number of times, for example 10 times.

In the domain of single-agent systems ESP has been shown to be very efficient so much that it has been extended to solve multi-agent learning tasks. Multi-Agent ESP [42] extends ESP to evolve a group of ANNs (agents). If ESP is parallelised this could potentially speed up the training of a group of agents in multi-agent tasks, that have real world application in domains such as gaming [54]. The next section details Multi-Agent ESP.

2.6.3 Multi-Agent ESP

Multi-agent problem solving involves several agents working together to achieve a common goal. Multi-Agent ESP extends ESP because ESP offers a solid foundation, as it has been shown to reliably solve problems faster and solve hard problems [23]. Each agent is controlled by an ESP-evolved ANN.

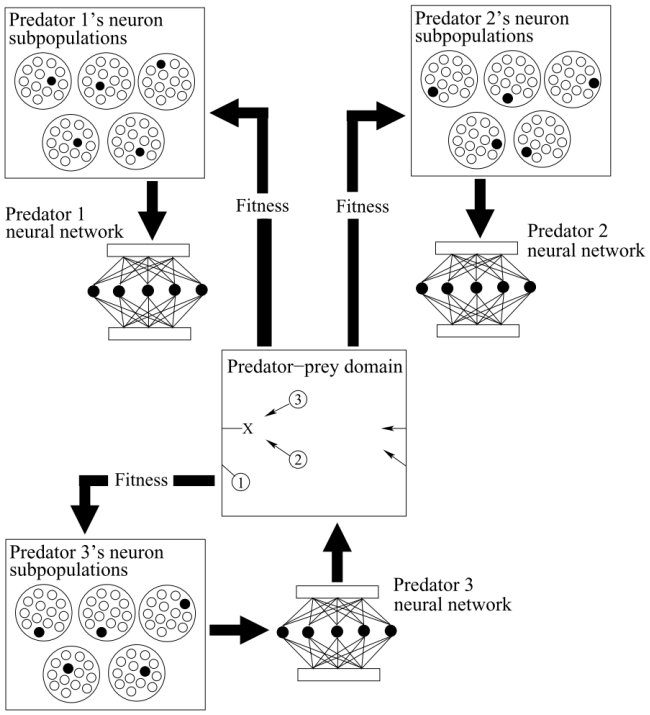


Figure 2.7: **The Multi-Agent ESP method**. Each team member is controlled by an autonomous agent that is evolved by the ESP method. The networks formed are feedforward networks, and are evolved in the prey-capture task. Figure taken from Yong and Miikkulainen [64].

In Multi-Agent ESP a team of neural networks is evolved using an Evolutionary Algorithm (EA) to solve a cooperative task. Yong and Miikkulainen [64] showed that evolving several autonomous, cooperating, non-communicating neural networks to control individual team members was more efficient and robust than evolving a single centralized controller to control the entire team. Parallelization of this method will yield great benefit to multi-agent systems. This work's focus is on the autonomous multi-agent approach described by Yong that involves following steps (Figure 2.7):

1. N autonomous agents are each controlled by their own network.
2. During an evolutionary cycle each network is formed using the ESP method.
3. The N networks are then evaluated together in the domain, as a team.
4. The fitness for the team (which is the average final distance from the prey) is then distributed among the neurons that contributed to the networks.

2.7 Parallelization of Cooperative Co-Evolution methods

In the field of CC the speed-up achieved from parallel implementations has been explored. Older implementations took the approach to adjust the synchronous algorithms to make them asynchronous [46, 10]. These implementations however did not truly leverage the power of multi-core architectures. In contrast to these approaches, some implementations have taken the multi-threaded approach [1]. One such implementation is a multi-threaded parallel version of ESP [21] on a shared-memory multi-processor machine, the 14-processor Sun Enterprise 5500. The implementation at the time took roughly a week to carry out the large simulations. However, that implementation was specific to that particular hardware architecture and therefore not easily portable. As possible future work, a client-server parallel architecture with a shared file system was proposed, where sub-populations of neurons are written to and read from. The clients would only synchronize after each generation and that there would be low input-output overhead, as most CPU time is spent at the evaluation phase. However this client-server architecture is not necessary with the current advancements of hardware. This work however also highlighted that the evaluation phase was the most time consuming and was therefore the focus of the parallelization.

More recently Nielsen et al. [46] proposed a framework for asynchronous Cooperative Co-Evolutionary Multi-Objective Algorithms (CCMOEAs). To evaluate the implementations, experiments were run on a dedicated node on an HPC cluster. The CCMOEAs used four sub-populations, each of them evolved in parallel in a dedicated thread on one of the four CPU cores used. This again had its limitations as it was specific to a particular hardware architecture.

To address computational overheads as complexity of a problem increases, what they termed, the "curse of dimensionality", Vu et al. [57] proposed a parallel Cooperative Co-Evolution Evolutionary Algorithm (CCEA). They suggested that for such problems decomposition was well suited, parallel processing in which each CPU executes a designated piece of work. They proposed a master-slave model for parallelizing the CCEA, where the evolution of populations resided on slaves, rather than just evaluating fitness, whilst the master behaves as a controller generating the populations and maintaining the best individual. The master then sends to each slave a population and the global best individual. In each slave a population is evolved using an Evolutionary Algorithm (EA) called Differential evolution (DE). After each evolution cycle, the slave sends back to the master the best individual as its representative for updating the global best. The number of slaves was limited as it caused bottleneck effects at the master processor.

CCEA was implemented using the MPICH2 library, a widely-used implementation of Message Passing Interface (MPI), using the C programming language. The setup included a population size of 100 and three computers each with two processors (six processors in total). The master was designated one processor. They however noted that using three slaves was slightly more efficient than using four slaves, which they attributed to inefficient thread management of MPI, which demonstrated the disadvantage of conventional multi-threaded programming. The use of a small population size and the small number of slaves also meant that the algorithm had

limited scalability. Vu's approach was also not as granular as the approach which focused on the parallelization of the evaluation phase, as it also included the evolution of populations at the slaves.

Optimization may be described as the process of solving a problem intelligently not laboriously [61]. This can be elaborated using the example of searching for a diamond in a forest, where the focus is on searching just the potential spots and not inch by inch. In a fairly recent survey, Mahdavi et al. [40] identified two approaches to solving large scale global optimization problems: cooperative co-evolution (CC) methods (decomposition methods) [48] and non-decomposition methods. The authors highlighted that the standard algorithms still suffer from one main deficiency; that the performance of the algorithms deteriorates when tackling high dimensional problems, as those have very large computational cost. They attributed this to the fact that, increasing the size of the problem dimension increases its search space, so in such cases an optimization algorithm needs to explore the entire search space efficiently, which is not a trivial task. The advantage of CC methods is that they use a divide and conquer approach, which decomposes such problems into multiple low dimensional problems, which are then potentially easier to solve. They highlighted some gaps in the state-of-the-art, such as optimal decomposition [7]. This pertains to design, grouping variables into some sub-components (near optimal decomposition) to significantly improve performance. Effort is required to fully develop decomposition methods with high performance and great accuracy on both non-separable (those that may need to be sequential or have dependencies) and separable components. Therefore, the design of decomposition methods is important. They also highlighted the need for more work to be done to understand CC characteristics (theoretical foundations), such as initial population and population size, as this affects the performance significantly [39, 35, 36]. They added that theoretical studies are limited. The application to real-world problems was also highlighted. There is a need to apply CC to real world challenging tasks. Scalability of large scale optimization problems was also highlighted as an essential requirement [65].

In another recent survey, the gaps in the state-of-the-art were further reinforced. Hussain et al. [34] highlighted some research gaps and future directions. They highlighted the need for well established and commonly agreed performance validation criteria, so as to establish firm conclusions about efficiency. They also reiterated on the need for theoretical and mathematical foundations, such as convergence analysis and exploration (probing of a much larger search space) versus exploitation (probing of a limited, but primary region) [11, 62]. Like the other surveys, they also highlighted scalability, that they described as, the ability to adapt, tune and evolve to large optimization problems with massive decision variables. Re-usability is another gap in the state-of-the-art. Boussad et al. [4] highlighted the need for a software framework that may help develop methods without building them from scratch.

Based on the aforementioned works and surveys, in order to reap the benefits of parallel hardware we propose a general parallel processing framework, one that focuses on optimal decomposition of the CC methods (spreading evaluations across multiple cores). The implementations derived from the framework should be able to run on any parallel computer (portable), highly

scalable, re-usable and not limited by a conventional multi-threaded approach.

3 Design and Implementation

The parallel processing framework that we propose is designed to parallelize any CC method. We aim to design a framework that breaks down the CC components into concurrent processes that are easily parallelizable. The following sections detail the design approach, the implementation details and the challenges encountered.

3.1 Approach

In order to effectively develop a parallel processing framework the serial algorithm needs to be fully understood and the concurrent tasks identified. The CCNE components are as follows: 1) initialization of the population, 2) creation of artificial neural networks (ANNs) from individuals in the population, 3) evaluation within the problem domain, and 4) recombination (evolution of the population of individuals). In theory two components can be carried out concurrently: the evaluation phase and the evolution of the individuals in the population, however from literature it has been established [22] that the most compute intensive task for Neuro-Evolution (NE) is the evaluation phase. The evaluation phase will therefore be the focus of our parallelization process. It has been highlighted that the evaluation phase contributed to 98% of CPU time [23]. The algorithms, for SANE, ESP and Multi-Agent ESP can be functionally decomposed, that is, 1) serial initialization of the population, 2) serial creation of ANNs from the individuals in the population, 3) parallel evaluation of the networks in the problem domain, then 4) serial evolution of the population of individuals. We can therefore in theory have a task-parallel processing framework, which is naturally suited to multi-core architectures. As part of our approach we considered four important aspects of parallel programs: *communication, synchronization, data dependencies and load balancing.*

We identified if any communication was required between processes and if any data was subject to being changed at the same time. We also needed to identify the scope of communications, that is, for example, does it involve two tasks; with one task acting as a sender/producer of data, and the other acting as the receiver/consumer. Also of importance is synchronization, that is, managing the sequence of work and tasks. We identified the most appropriate synchronization approach and the synchronization points. Another important aspect that we considered was data dependencies. A dependence exists between program statements when the order of statement execution affects the results of the program. Data dependencies can inhibit parallelism, so we identified which parts of the algorithms had data dependencies. We also considered load balancing, which is the distribution of approximately equal amounts of work to the core processors. It is vital to parallel programs for performance reasons and we ensured that we factored it into our design. We achieved load balancing by equally partitioning the work each core receives, therefore each core processed the same number of evaluations.

3.2 A Parallel Processing Framework

Our parallel processing framework is shown in Figure 3.1. This shows one generation of the evolutionary process, and details the evaluation phase that occurs on each processor core. The

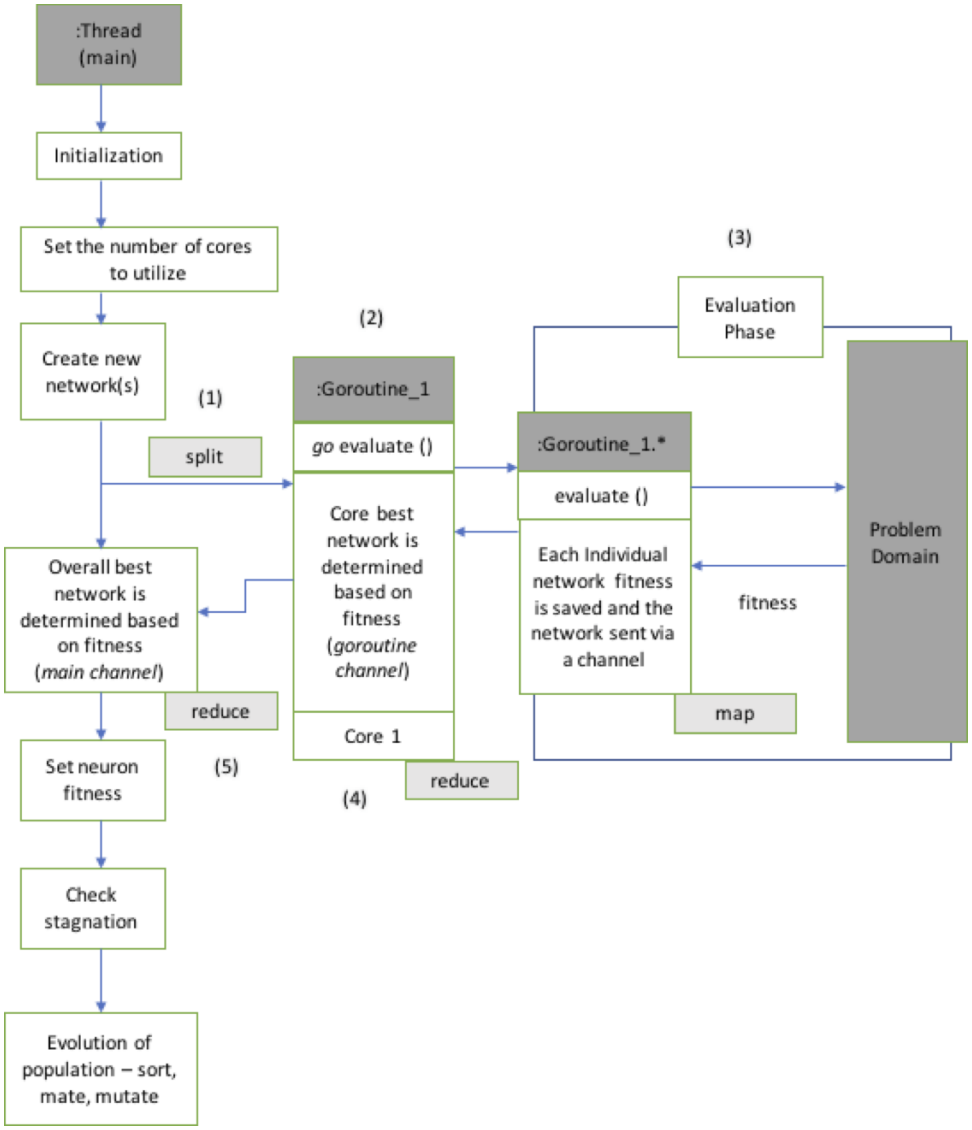


Figure 3.1: **A Parallel Processing Framework for Cooperative Co-Evolution methods**. A goroutine is a lightweight thread, generated by the Go runtime. Each core goroutine spawns multiple goroutines. Channels are the communication conduits that are used for synchronization and sharing of data between goroutines.

focus of our work begins at Step 1; load balancing. This is where the networks to be evaluated are evenly distributed across multiple cores. Each split, a number of networks, is processed in its own goroutine (a light-weight thread generated by the Go runtime), for example if we have to carry out 20 evaluations, that means 20 networks are created and evaluated, therefore the split will be as follows:

$$[\text{n1} \ldots \text{n20}] \ \textit{split} \ [[\text{n1} \ldots \text{n5}], [\text{n6} \ldots \text{n10}], [\text{n11} \ldots \text{n15}], [\text{n16} \ldots \text{n20}]]$$

If the total number of cores is four, then each core goroutine receives five networks to evaluate as shown in Step 2. Each core goroutine then spawns separate goroutines to evaluate each

network in the task environment (Step 2). Step 3 represents the evaluation phase, the spawned goroutines run the actual *evaluate* method. The fitness of a particular network is determined on its spawned goroutine and the network (and its respective fitness) is sent via a channel to the core goroutine. The channel represents the communication aspect of our process. This is the producer or receiver pattern where the main goroutine on the core behaves as a receiver and the spawned goroutines behave as producers. The evaluate method performs a map operation, mapping a network to a network and fitness (key-value pair), since after evaluation a fitness value is set for each participating network:

[network] *evaluate* [network, fitness], for example:

[n1] *map* [n1, 10]
[n5] *map* [n5, 5]

The goroutine in (Step 2) then receives all networks from the spawned goroutines via a channel and determines the best core network using a reduce operation:

[[n1, 10] ... [n5, 5]] *reduce* [n1, 10] (core best)

A barrier approach to synchronization is employed at this phase, as the channel blocks until it receives all the data from the spawned goroutines. This is our first synchronization point. The list of network-fitness key-value pairs is reduced via an operation that determines the network with largest fitness value. Each core goroutine then sends their best ANNs via a channel to the main thread, which then determines the overall best network via another reduce operation. Given for example that *n18* is the core best on core four, and *n1* is the best network on core one, the operation would be as follows:

[[n1, 10], ... [n18, 8]] *reduce* [n1, 10] (overall best)

The overall best is again the network with the largest fitness value. The example just described outlines the process followed by MapReduce. MapReduce is a programming model that has its roots in functional programming. It consists of two functions, *map* and *reduce*. Map operates on a list of values producing a new list of values. It does this by applying the same computation to each value. The reduce operation takes a list of values and collapses or combines those values into a single value (or a number of values), again by applying the same computation to each value. This model has been proven to be particularly valuable for large datasets [13]. This model works well on parallel hardware, where data items can be split across many processors or processor cores.

3.3 Challenges

Mutual exclusion is one common problem whenever writing concurrent algorithms. It should always be clear which variables are shared and which ones are local. As part of our approach, data dependencies were considered, however during the early phase of the parallel implementation, the neurons that participated in the evaluation phase were erroneously updated in the evaluation phase (Step 3 of Figure 3.1). This is incorrect because a neuron may be selected to form a number ANNs and there is the risk of race conditions if several goroutines update a neuron's fitness at the same time. This error resulted in the evolution time taking longer than expected. To remedy this issue we made that particular operation a critical section. A critical section is a code segment that needs to be executed atomically. The neuron fitness updates were carried out serially on the main thread, after the evaluation phase. This ensured that our data remained in a consistent state.

Another common problem is the producer-consumer problem. In this scenario a process or processes generate data and the consumer receives it, for further computation. This is the model we employ for communicating between goroutines, data is sent via a channel and a reduce operation is carried out by the consuming-goroutine. According to Axford [2], for this model to be acceptable: 1) the items must be received by the receiving-goroutine in the same order as they were sent by the producing-goroutine, 2) no data must be lost in transit, and 3) there should be no tampering of the data in transit. Also the mechanism for passing the data (the channel) should itself employ some synchronization constraints on the producer-goroutine and consumer-goroutine, for example, the consumer cannot receive until the producer sends or the channel must hold data until a consumer is ready to receive it. In our parallel processing framework we have a multiple-producers-and-one-consumer model. Go's first class primitives made this issue easy to solve, as the author simply used Go channels that adhere to the rule-set described by Axford.

Another challenge is how to identify blocking operations. During the early phase of the implementation we enabled profiling [38] for all our implementations. As a result we discovered the use of the global rand function was causing a blocking operation. The rand function is used for initializing our neuron synaptic weights. This had an impact on the speed of the evolutionary process. To fix this we updated our code to use a non-global, non-blocking rand function.

4 Validation

Validation aims to show that the implemented CCNE methods are correct. It is of particular importance as the original case study methods [44, 22, 64] were implemented many years ago and their source code is difficult to run on modern architectures. Our approach entails unit testing the software modules for low level confirmation and then higher level confirmation of the new serial and parallel versions in two validation tasks: double pole balancing and prey-capture. Double pole balancing is used to validate the single-agent methods, SANE and ESP, whereas the prey-capture task is used to validate the multi-agent method, Multi-Agent ESP.

The pole balancing task offers a good procedure for validation as it is analogous to the real world and is easy to understand. When visualised the trained artificial neural network (ANN) can be observed performing the task and based on the constraints of the simulation model, the failure angle and track size, it is easy to determine whether the controller is actually performing the task correctly. The same applies to the prey-capture task, as the prey being successfully captured can be visualised. The validation results were based on whether or not the output was within the defined acceptable bounds, and whether or not the visualization demonstrated that the task was completed successfully.

In Go, packages are system directories that are used to organize source files. The package ecosystem in Go encourages code reuse and modularity [38], this makes it possible to unit test at the package level. Unit testing validates that each individual package works correctly, whereas validation of the serial versions that the Go serial versions implemented work as correctly as the original implementations. Validation of the parallel versions confirms that the parallel versions are still correct after the parallel processing framework has been applied to the serial implementations. Finally the visualizations provide a real world model that affirms that the task is being performed correctly.

4.1 Unit Testing

Unit testing is a software development practice in which the parts of a program, called units, are independently tested to determine correct operation [12]. This process can be carried out manually but is often automated, that is, by writing test code and ensuring that running the tests is repeatable. Go provides support for automated testing through its testing package. In our study we treated each core package as a unit and wrote unit tests for those packages. Breaking up a program into packages, allows the individual units to be reusable within any CCNE method, with the only requirement being the adjustment to the parallel algorithm implementation where required. In CCNE methods, neurons are segregated into multiple populations and an ANN is formed by selecting neurons from the populations. The network is then evaluated within a problem domain (environment). This formed the basis for selecting which core packages needed to be unit tested in this study, therefore neuron, population, network and environment packages were tested.

The neuron package consists of a neuron source file and the corresponding unit test file. We

tested the neuron methods and this entailed; testing the initialization of a new neuron, the creation of a new set of synaptic weights for a neuron and the setting of a new fitness value for a neuron. To test the initialization of a new neuron, we tested whether after initialization the neuron had the expected synaptic weight size. We did this by comparing the size value specified at initialization to the neuron weight size after initialization, the weight values are set to zero at initialization. To test the creation of the synaptic weights we compared the weight values after creation to zero, to ensure that no weight values were zero. The setting of the fitness was tested by setting the fitness for a single neuron a number of times and confirming that the fitness thereafter matched the expected cumulative fitness, for example, if we set the first fitness value to one, and then to one again, we expect a cumulative fitness of two.

The population package consists of a package source file and the corresponding unit test file. For the population package the creation of a new population is tested. We did this by confirming that the number of individuals (neurons) in the population after creation matched the size value specified on creation. The network package is tested for two network types, feedforward and recurrent. The main test is the creation of a new ANN, and additional tests confirm that the correct gene size is returned. The calculation of the gene size differs depending on the network type. This is because for recurrent networks the genesize is the sum of the number of inputs and the number of hidden units, whereas for feedforward networks the genesize is the sum of the number of inputs and the number of outputs. The total inputs are verified, as the inputs are based on whether there is a bias or not. The presence of a bias means an extra input.

The environment package is not necessarily a core package, as one may implement multiple environments to evaluate a CC method. However due to the great deal of calculations carried out in the double pole balancing task, the task was tested. The creation of a new cart-pole environment is tested, as well as the initialization of the state variables. Finally a force being applied to the cart is tested, to ensure that it is not zero. This is because this can be a source of error for the pole balancing task. For this reason, the force applied was restricted to be no less than $\pm 1/256 \times 10$ Newtons so that the controllers cannot maintain the system in an unstable equilibrium by outputting a force of zero when the poles are vertical [23]. The unit tests are available online[1].

4.2 Methods for validation tasks

We began by determining the necessary input parameters for the validation tasks, based on literature [23, 64]. Thereafter the programs (serial and parallel) were run in the validation tasks and the output observed and captured into a file upon successful completion of the tasks. The output files were then analysed to ensure that the output was within the correct bounds for each task.

For the double pole balancing task, failure is based on two parameters: the position of the cart within the task environment (the track) and the angles of the poles from the vertical. The

[1] `https://github.com/edmore/cooperative-coevolution`

cart balances the poles (long pole length one metre and short pole length 0.1 metres) on a 4.8 metre track, and the position of the cart is measured from the centre of the track, therefore any position in the range [-240, 240] centimetres is valid. For the poles, angles outside the range [-36, 36] degrees represent failure.

The analysis of the validation output for the prey-capture task was based on whether or not the coordinates of the predators and prey were within the toroidal (task) environment and whether on capture, at least one of the predators was able to occupy the same position as the prey. The output files were then used as input to visualization programs, to confirm that the implementation behaved correctly, that is, the poles were balanced for the simulated time period and the predators were able to successfully capture the prey. Our implementations use a *sim* parameter, that when set to true, enables the capturing of successful states into an output file. These states can then be replayed using the double pole balancing visualizer (Figure B.1) and the prey-capture visualizer (Figure B.2). An additional parameter, *markov*, was used for the new implementations of the double pole balancing task to indicate whether a task was non-Markov or Markov. The parameter when set to false initializes the task as a double pole balancing task with incomplete state information (non-Markov); it is set to true by default.

Since SANE is the precursor to ESP it was implemented at a time when the single pole balancing task was deemed adequate. Moriarty and Miikkulainen [44] used the single pole balancing task for his work, however for this work our *neuron* SANE implementation was validated on the double pole balancing task with complete state information. The validation output was then replayed using the double pole balancing visualization program. The program plays back the state variables for each time step resulting in a visual view of the controller balancing the poles. Gomez [23] details the parameters used for the pole balancing comparisons in the original implementation. Part of our validation involved ensuring that the new implementations accepted all the parameters he used and generated errors on invalid input. We used the same parameters Gomez used to validate our work. These double pole balancing task parameters, are shown in Table 4.1.

	SANE (Markov)	ESP (Markov)	ESP (non-Markov)
h	5	5	5
n	40	40	40
b	-	10	10
markov	true	true	false
spl	0.1 metres	0.1 metres	0.1 metres
*s*im	true	true	true

Table 4.1: **The parameters for validating SANE and ESP in the double pole balancing task**. ESP was validated in both the Markov and non-Markov versions of the task. *h* represents the number of hidden units, *n* the number of individuals in a population, *b* the burst mutation, *markov* whether or not it is a Markov task, *spl* the short pole length in metres and *sim* denotes whether or not we capture the states for playback in the visualization program.

The prey-capture task was used to validate the Multi-Agent ESP method, the parameters used are shown in Table 4.2. The prey-capture visualization program was then used to playback the states of the predators as they captured the prey. The default parameters used by Yong [64] were used, that is, one prey and three predators. The prey moves as fast as the predators and always away from the nearest predator, meaning one predator cannot catch the prey. The prey was started at coordinates [16,50] of a 100 x 100 square toroid and the predators were positioned at the bottom left corner. The prey was started at the same position for all simulations so that we could validate the serial and parallel versions from a single point of reference. An additional validation task was performed where only one predator was tasked with catching the prey. This was used to confirm that one prey would be unable to catch the prey. This was done by setting a flag *maxGens* to limit the number of generations the controller (predator) was evolved and also the *pred* flag was set to one so that only one predator was configured to capture prey.

Ten simulations were run to validate each implementation, however since the state variables differ for each simulation the results in the next section show the analysis of only one of the simulations, for each implementation.

	Multi-Agent ESP
h	10
n	100
b	200
pred	3
prey	1
trialsPerEval	9
sim	true

Table 4.2: **The parameters for validating Multi-Agent ESP in the prey-capture task**. *h* represents the number of hidden units, *n* the number of individuals in a population, *b* the burst mutation, *pred* the number of predators, *prey* the number of prey (one by default), *trialsPerEval* the number of trials per evaluation and *sim* denotes whether or not we capture the states for playback in the visualization program.

4.3 Results

The following section details the results of the NE methods: SANE, ESP and Multi-Agent ESP in the validation tasks. Simulations were run on a single parallel computer running Mac OS (Table A.1) and 100 time steps represents approximately 30 minutes of simulated time.

4.3.1 SANE

The position of the cart and the angles of the poles were within the acceptable bounds (detailed in the methods section) for both the serial and parallel versions for the duration of the simulation, as shown in Figure 4.1. For the serial version of SANE, the position of the cart is maintained in the [30,40] centimetres range for most of the simulation (Figure 4.1a), which is within the [-240, 240] centimetres acceptable range and the angles of the poles are within the [-36, 36] degrees acceptable range. The results also demonstrate the ease at which the poles are

balanced, the pole angles are maintained within the [-2, 2] degrees range (Figure 4.1b), meaning that they are upright on the cart. This is due to the simplicity of the double pole balancing task with complete state information.

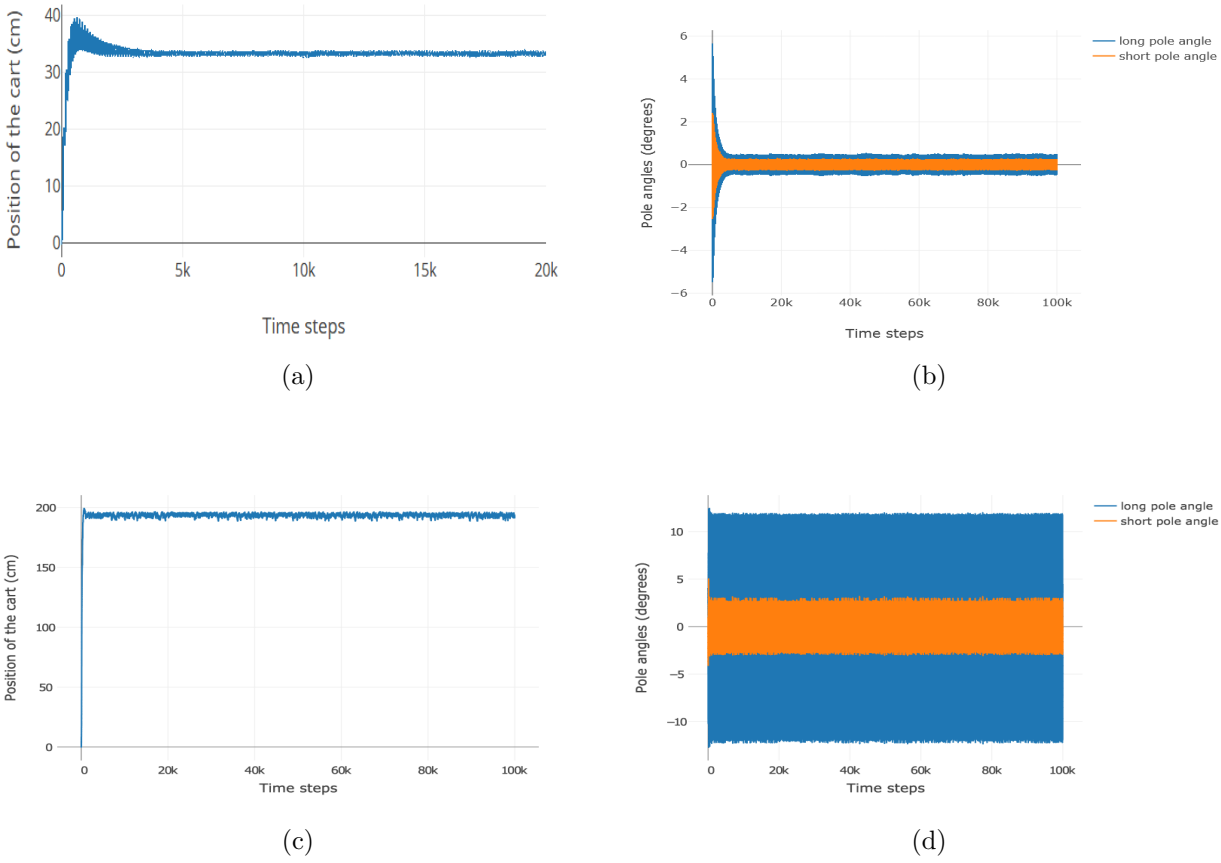


Figure 4.1: **Balancing two poles with complete state information**. SANE is shown to balance the poles within the prescribed bounds for the entire simulation. a) The position of the cart over 100,000 time steps for the serial version of SANE. b) The angles of the short pole and long pole over 100,000 time steps for the serial version of SANE. c) The position of the cart over 100,000 time steps for the parallel version of SANE. d) The angles of the short pole and long pole over 100,000 time steps for the parallel version of SANE.

The results of the parallel version of SANE also show that the task parameters are within the acceptable bounds. The position of the cart is maintained at approximately 200 centimetres, from the centre of the task environment (Figure 4.1c) and the pole angles are within the [-36,36] degrees range (Figure 4.1d). The last stage of validation involved replaying the states in the visualization program. The visualizations of SANE serial and SANE parallel in the pole balancing task also demonstrated the implementations correctly balancing the poles.

4.3.2 ESP

ESP was validated in the pole balancing task with complete (Figure 4.2) and incomplete (Figure 4.3) state information. The results show that for the Markov version of the double pole

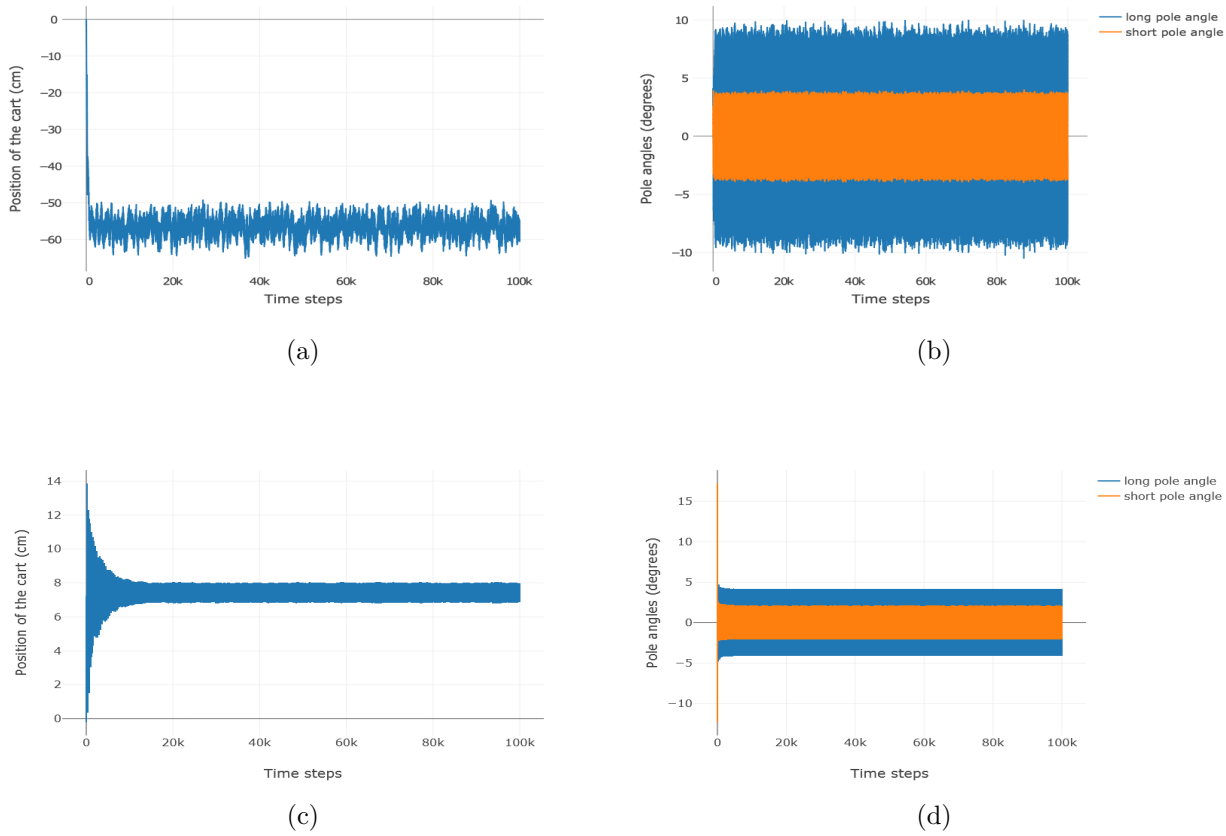


Figure 4.2: **Balancing two poles with complete state information**. ESP is shown to balance the poles within the prescribed bounds for the entire duration of the simulation a) The position of the cart over 100,000 time steps for the serial version of ESP. b) The angles of the short pole and long pole over 100,000 time steps for the serial version of ESP. c) The position of the cart over 100,000 time steps for the parallel version of ESP. d) The angles of the short pole and long pole over 100,000 time steps for the parallel version of ESP.

balancing task, the state variables are within the prescribed bounds for the entire simulation. We observe the serial version (Figure 4.2a and 4.2b) balance the poles more erratically than the parallel version (Figure 4.2c and 4.2d), but still within acceptable bounds. The parallel version maintains the poles within the [-5, 5] degrees range for most of the simulation. This means that the poles are vertical to the cart for most of the simulation.

For the non-Markov version of the task we observe that both versions balance the poles more erratically but within the acceptable constraints. For the serial version (Figure 4.3b) the cart position is close to the centre of the track for most of the simulation and for the parallel version (Figure 4.3d), the cart position is more erratic (ranging between approximately [20, 160] centimetres for the duration of the simulation). The movement of the cart for the parallel version demonstrates the complexity of the non-Markov task over the Markov version, as the poles are not balanced to stability easily. The angles of the poles for the for the serial (Figure 4.3a) and parallel (Figure 4.3c) versions are also within the acceptable bounds for both implementations. The visualizations of ESP in the Markov and non-Markov tasks also confirmed the poles being

balanced correctly.

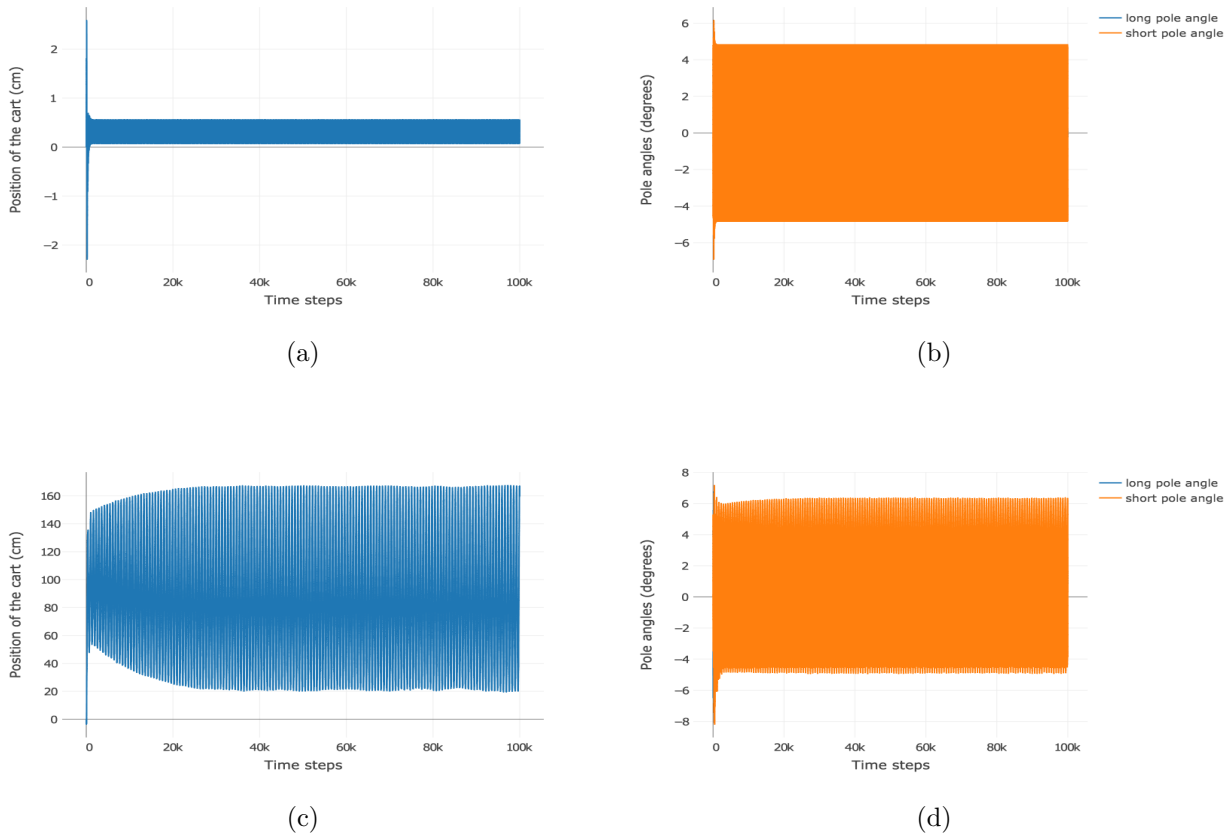
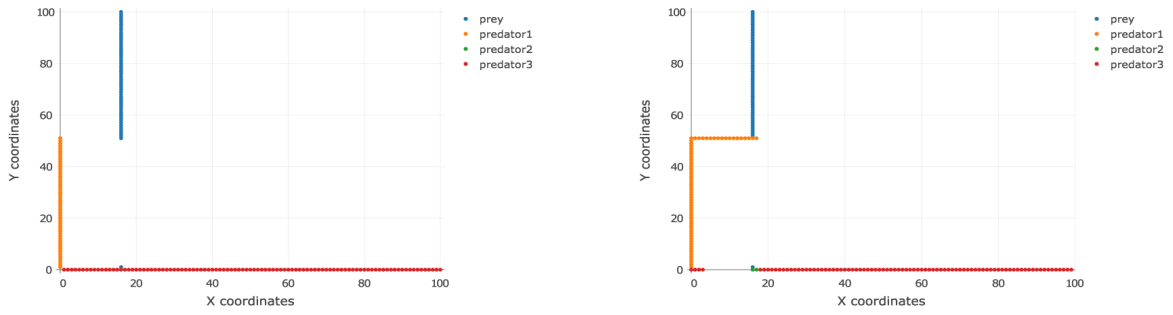


Figure 4.3: **Balancing two poles with incomplete state information (non-Markov)**. ESP is shown to balance the poles within the prescribed bounds for the harder version of the double pole balancing task. a) The position of the cart over 100,000 time steps for the serial version of ESP. b) The angles of the short pole and long pole over 100,000 time steps for the serial version of ESP. c) The position of the cart over 100,000 time steps for the parallel version of ESP. d) The angles of the short pole and long pole over 100,000 time steps for the parallel version of ESP.

4.3.3 Multi-Agent ESP

The results of the validation of the serial and parallel versions are shown in Figure 4.4. The results show that the prey and predator movements are valid, that is, the coordinates for the predators and prey remain within the toroidal environment and catches are observed by a predator occupying the same position as the prey. In Figure 4.4a we observe the movements of the predators and the prey for the serial version. The agents (predators and prey) remain within the simulation environment and the prey-capture is demonstrated by the predator occupying the same grid cell as the prey. Similarly for the parallel version Figure 4.4b, the agents remain within the simulation environment and the prey is correctly captured.

To validate whether or not one predator could catch prey, the terminal output as the program



(a) (b)

Figure 4.4: **The prey-capture task**. Multi-Agent ESP is evolved in the prey-capture task and we observe the predators capture the prey by occupying the same grid cell as the prey. a) The coordinates of the prey and predators in the prey-capture task for the serial version of Multi-Agent ESP method. In the above illustration predator3 catches the prey. b) The coordinates of the prey and predators in the prey-capture task for the parallel version of Multi-Agent ESP method. In the above illustration predator2 catches the prey.

ran had to be observed, as the states were never captured in the output file because for each generation there were zero catches. This confirmed that the prey always moves away from the nearest predator, and since they move at the same speed it can never catch the prey.

4.4 Summary

The analysis of the validation output and the visualizations of the implemented NE methods: SANE, ESP and Multi-Agent ESP, gives us confidence that they are implemented correctly. Their state variables all fall within the prescribed bounds of the validation tasks. Visualizations mentioned in this chapter are available online[2]. Additional validation experiments were carried out on the HPC cluster and their timings are available in Appendix C. In the next chapter the NE methods are benchmarked to establish the performance gains due to the application of the parallel processing framework.

[2] `https://github.com/edmore/cooperative-coevolution`

5 Benchmarking

In this chapter we describe the experimental setup used for benchmarking the three parallelized Neuro-Evolution (NE) methods: SANE, ESP and Multi-Agent ESP, and thereafter we discuss the performance of the implementations. Benchmarking a parallel program demonstrates the performance gains that parallelisation introduces to an initially serial program. Although the double pole balancing task and the prey-capture task offer a good basis for validating the CCNE methods, they also offer a good basis for benchmarking them. This is because these tasks are easily configurable, tasks of varying complexity can be easily set up and metrics (such as CPU runtime) can be gathered after each run. The double pole balancing task with incomplete state information can be made more complex, for example, by gradually increasing the short pole length.

The evaluation phase is the most computationally intensive phase for SANE, ESP and Multi-Agent ESP and therefore the benchmarking was structured in such a manner as to demonstrate the effect that the evaluation phase has on runtime, and the performance gains due to the application of the parallel processing framework. Benchmarking was carried out by evaluating the speed-up attained when the NE methods were evaluated for an increasing number of cores. In the context of this study, runtime refers to the CPU time taken for the artificial neural network (ANN) or team of ANNs to be evolved in the benchmark tasks. The runtime is used to calculate the speed-up, which is the ratio of the parallel to the sequential runtime. The ideal speed-up for N cores is N speed-up. We expected a linear speed-up, as the number of cores was increased. The SANE method implemented in this work is a single population NE method, referred to as neuron SANE [44]; the algorithm lends itself naturally to the parallel processing framework and was benchmarked to demonstrate the general nature of the parallel framework and to provide a basis for gauging the efficiency of CCNE methods over NE-only methods.

5.1 Experimental Setup

The following sections detail the experimental setup used to benchmark the NE methods. The NE methods were all benchmarked on a Linux High Performance Computing (HPC) cluster (Figure A.2). Each worker node utilized in the cluster had 64 cores, but our experiments used up to 60 cores. The number of cores utilized for each experiment ensured that the evaluations were evenly distributed across the available cores.

5.1.1 Pole Balancing Experiments

Drawing on the work of Gomez [21], the simulation environment for the double pole balancing task in this study was implemented using a realistic physical model with friction, and fourth-order Runge-Kutta integration with a step size of 0.01 seconds. The pole balancing task is a series of cycles that begins when the ANN interacts with the simulation environment by receiving state variables as inputs and then a non-zero force is output by the ANN, which is then applied to the cart. At each time step (0.02 seconds of simulated time), the ANN receives the new state variables. The cycle is repeated until a failure condition is reached, for example

the poles fall or the cart goes off the defined track. A task is considered solved if the controller can keep both poles within the specified angle range of [-36, 36] degrees from the vertical and the cart between the specified bounds of the track (2.4 metres either side of the middle of the track), and this should be done for 100,000 time steps, which is equal to over 30 minutes of simulated time.

The primary metric, *speed-up*, was evaluated by increasing the number of cores and observing whether the ANN was evolved faster, the more cores were added. One hundred simulations were carried out for each number of cores, for tasks of varying complexity (short pole lengths 0.6, 0.7, 0.8 and 0.85 metres) and the results of each experiment averaged. All simulations were successful, however only a small percentage (approximately three percent) were outliers, and therefore discarded. The length of the long pole is one metre. The task parameters used for SANE and ESP are shown in Table 5.1. The difference was that SANE was only benchmarked in the pole balancing task with complete state information, as it can only evolve feedforward ANNs (Figure 5.1b) , whereas ESP was benchmarked in both variations of the double pole balancing task. A fully recurrent ANN (Figure 5.1a) was evolved using the ESP method, as feed-forward networks are unable to solve the non-Markov version of the double pole balancing task.

(a) (b)

Figure 5.1: **Artificial Neural Network (ANN) types and their state variables for the double pole balancing task**. a) An example of a recurrent ANN. This network type was used to benchmark ESP in the non-Markov version of the double pole balancing task. b) An example of a feedforward ANN. This network type was used to benchmark SANE and ESP in the Markov version of the double pole balancing task. Figure taken from Gomez [21].

hidden units (h)	5
individuals in the subpopulation (n)	120
CPU cores utilised (CPUs)	[1, 2, 4, 8, 12, 15, 20, 24, 30, 40, 50, 60]
length of short pole(spl)	[0.6, 0.7, 0.8, 0.85] metres

Table 5.1: **The double pole balancing task parameters**. The parameters were used for both the Markov (for SANE and ESP) and non-Markov (for ESP) tasks.

5.1.2 Prey-capture Experiments

Drawing on the work of Yong and Miikkulainen [64] the simulation environment consists of three or more predators in a 100 × 100 toroidal environment, with no obstacles or barriers.

Each predator is autonomous (controlled by its own ANN) and does not communicate with the other predators. The predators always start at the bottom-left corner of the environment and the environment is stochastic only in the prey's starting location. The prey capture task is a series of capture attempts that are deemed successful when the prey is caught. In our variant of the task the prey is caught when the predator moves into the same grid cell as the prey. All agents can move in four directions: north, south, east and west. Failure is when the predators have not caught the prey in 150 moves. The best team of predators captures the prey in the minimum number of moves. Also, the prey moves at the same speed as the predators, and always away from the nearest predator and because of these constraints it is impossible to catch the prey without cooperation [64]. What makes the prey-capture task such a challenge is that the predators are co-evolved with no communication and the prey moves at the same speed as the predators. A series of experiments were conducted to determine the effect of applying the CC parallel processing framework to the Multi-agent ESP method. 100 simulations were performed to evaluate each benchmarked number of cores. The number of predators was also varied, meaning one set of simulations was carried out with three predators and another set with six. Table 5.2 shows the task parameters used for the task.

There were some variations to Yong and Miikkulainen's [64] setup to ensure that the tasks demonstrated the effect that evaluations have on evolution time. Since the emphasis was on reducing the time taken during the evaluation phase, no incremental evolution [23] was applied, the neural networks were not gradually evolved, by steadily increasing the task complexity, they were exposed to the hard task from the very beginning. Complexity of this task can be varied by adding more individuals to the population, varying the number of predators or by increasing the number of trials performed per evaluation. Also burst mutation [23] was set to only occur after 200 generations. Each evaluation is a series of trials, after each evaluation the cumulative fitness is averaged. For each evaluation cycle there were 5,400 evaluations (number of individuals x 10) and for each evaluation, nine trials. This equates to 48,600 evaluations (team evaluations x trials) in total. A large population size was used (540 neurons) to increase the complexity of the task.

To perform the nine trials per evaluation, the toroidal simulation environment was split up into nine equal squares (Figure 5.2) and the prey was started at the centre of each of the squares, for each trial. The squares allow for the sampling of different starting positions [64].
A team of predators was only deemed to be successful if they had a 100% catch rate, that is, caught the prey for every single evaluation.

5.2 Results and Discussion

We first discuss the results of our double pole balancing experiments and then we discuss the results of the prey-capture experiments.

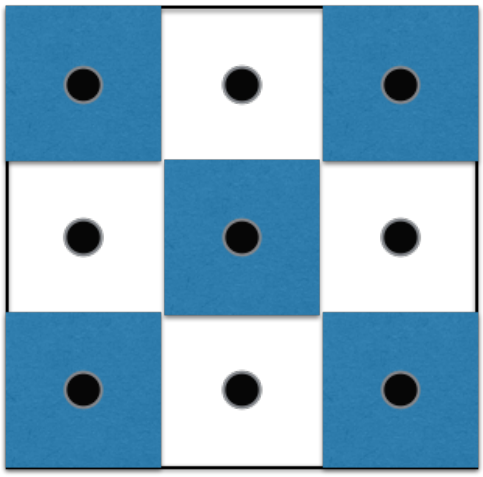


Figure 5.2: **Prey-capture trials per evaluation**. The simulation environment is split up into nine equal squares. Each black dot represents a possible starting position for the prey in each trial, that is, for each trial the prey is started at only one of those positions. The fitness scores for the trials are averaged to give the fitness score for the evaluation.

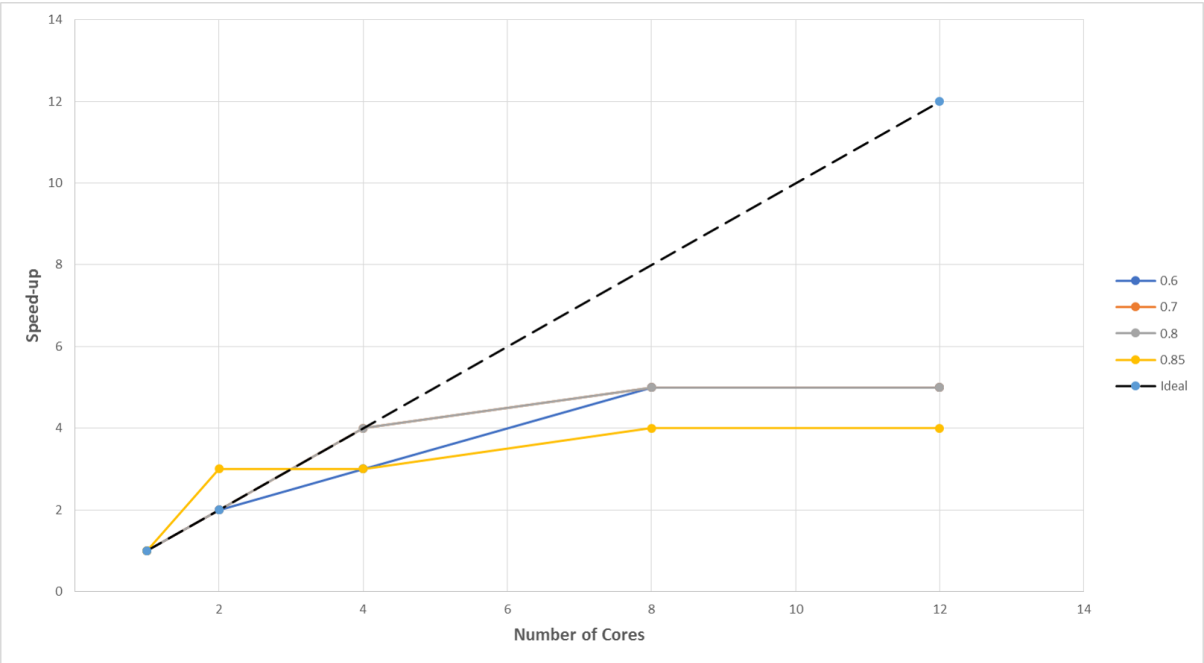
preds	3	6
n	540	540
h	15	15
CPUs	[1, 2, 4, 8, 12, 18, 24, 36, 45, 54, 60]	[1, 2, 4, 8, 12, 18, 24, 36, 45, 54, 60]
trialsPerEval	9	9

Table 5.2: **The prey-capture task parameters**. *preds* represents the number of predators. n the number of individuals in a population. h the number of hidden units. *CPUs* the number of CPU cores utilized and *trialsPerEval* the number of trials per evaluation.

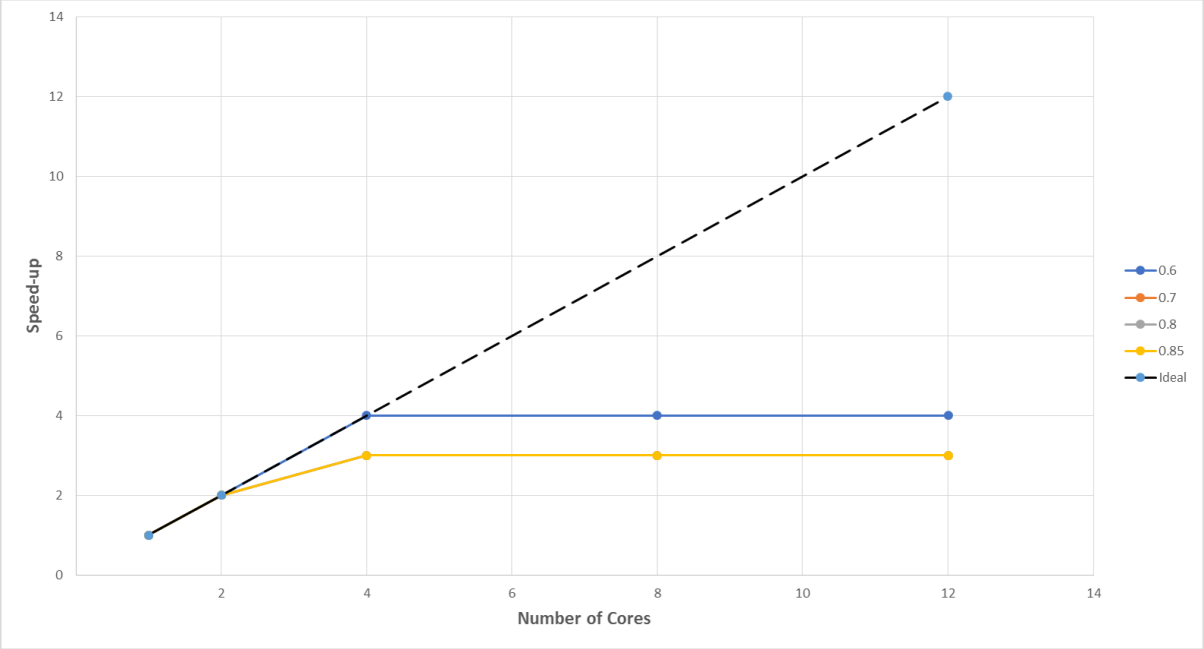
5.2.1 Pole Balancing Comparisons

Figure 5.3 shows the speed-up achieved by SANE and ESP in the double pole balancing task with complete state information (Markov). The speed-up of SANE is shown in Figure 5.3a. The maximum speed-up achieved by SANE is 5x for the short pole lengths 0.6, 0.7 and 0.8 metres. We observe that the Markov version of the task gets simpler the closer in length the short pole is to the long pole, therefore it is not surprising that the maximum speed-up is with the harder versions of the task. However, at two cores there is a speed-up of 3x for the easiest task (short pole length of 0.85 metres). This spike in speed-up is most likely due to an inferior serial implementation and simplicity of the task. At two cores the network evaluations run in parallel on two cores and because there is very little communication overhead and the task is simple, the evolution time is much smaller.

There is a linear speed-up for the short pole lengths of 0.7 and 0.8 metres for the range of cores up to four cores. These variations of the task are harder than the version with a short pole length of 0.85 metres. This indicates that for harder versions of the task we get a linear speed-up and we can possibly expect the same for more complex tasks (such as ones with larger



(a)



(b)

Figure 5.3: **Balancing two poles with complete state information**. The graphs show the speed-up for up to 12 cores, as the speed-up stagnates thereafter. a) SANE efficiently solves the task for a varying number of cores, for the short pole lengths 0.6 (blue), 0.7 (red), 0.8 (grey) and 0.85 (yellow). The broken line represents the ideal speedup. b) ESP efficiently solves the task for a varying number of cores, for the short pole lengths 0.6 (blue), 0.7 (red), 0.8 (grey) and 0.85 (yellow). The broken line represents the ideal speedup.

population sizes) on more cores. An interesting observation is that the speed-up stagnates at 8 cores. This indicates the simplicity of the Markov version of the double pole balancing task, because even given the increase in the number of cores and the overhead expected due to the

goroutine (a Go light-weight thread) communication, there is no speed-up after 8 cores.

We also observe a maximum speed-up of 5x when ESP is benchmarked in the Markov version of the double pole balancing task (Figure 5.3b), for the short pole lengths of 0.6, 0.7 and 0.8 metres. Like SANE, the lower speed-up was observed for the simpler variant of the task, short pole length of 0.85 metres. There is also a steady speed-up then stagnation for ESP. However ESP stagnates at four cores, relative to SANE's stagnation at eight cores, meaning that ESP achieves a maximum speed-up faster than SANE. ESP's stagnation can also be attributed to the simplicity of the Markov version of the double pole balancing task. We observe a linear speed-up for the harder versions of the task (short pole lengths 0.6, 0.7 and 0.8 metres) up to four cores. This again indicates that we can expect a linear speed-up for harder tasks on more cores.

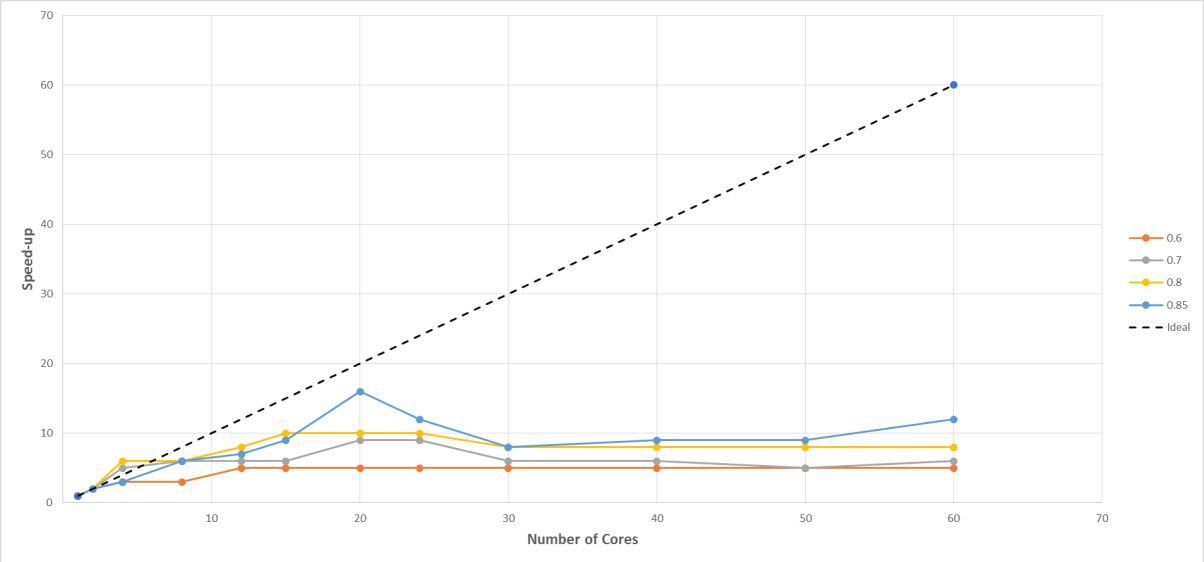


Figure 5.4: **Balancing two poles with incomplete state information (non-Markov).** The speed-up of the parallel ESP implementation in the non-Markov version of the task. ESP efficiently solves the task for a varying number of cores, for the short pole lengths 0.6 (red), 0.7 (grey), 0.8 (yellow) and 0.85 (blue). The broken line represents the ideal speedup.

Figure 5.4 shows the speed-up of ESP in a harder variation of the pole balancing task, the non-Markov version (with no velocity information). We observe that the task gets harder the closer to the long pole the short pole length is, which is the reverse of the observation with the Markov version of the task. The hardest version of the task benefits from more cores, a maximum speed-up of 16x is achieved at 20 cores for the short pole length of 0.85 metres. We also observe linear speed-up (with an R-squared value of 0.94) for the range of cores up to 20 cores for the short-pole length of 0.85 metres. The R-squared value was calculated using Microsoft Excel. A linear regression line (trend-line) was added to the data in the graphs and the R-squared value displayed on the chart. The closer to 1.0, the better the fit of the regression line. This indicates that the parallel processing framework is likely to benefit more complex tasks (such as ones with larger population sizes) on more cores. For all versions of

the task we observe a speed-up and then a reduction in the speed-up. Of particular interest is the steep reduction in speed-up for the hardest version of the task, and this can be attributed to high communication overhead [45] over 20 cores and that the task is not complex enough to have performance gains over 20 cores. We also observe a spike in the speed-up for the simpler versions of the task (short pole lengths 0.6, 0.7 and 0.8 metres), at four cores. This is likely due to a inferior serial implementation. There is however an unexpected speed-up for the short pole lengths 0.7 metres and 0.85 metres at 60 cores and the reason for this is unclear, as we would expect greater communication overheard and thus either a stagnant speed-up or a reduced speed-up. The results demonstrate the complexity of the non-Markov version of the task (in comparison to the Markov version), as it takes a longer runtime to evolve the ANNs in the harder benchmark task. We furthermore achieve a better speed-up in the more complex task which demonstrates the suitability of the parallel implementations in solving larger scale problems.

5.2.2 Prey-capture Comparisons

Figure 5.5 shows the results of benchmarking Multi-Agent ESP in the prey-capture task. With three predators we observe a maximum speed-up of 13x at 45 cores. We observe an expected decline at 54 cores, due to scheduling and communication overheads [45], and the speed-up stagnates from there onwards. At the maximum speed-up at 45 cores, the runtime (the time taken to evolve the predators to successfully complete the prey-capture task) for the serial version is 13 minutes versus approximately one minute for the parallel version.

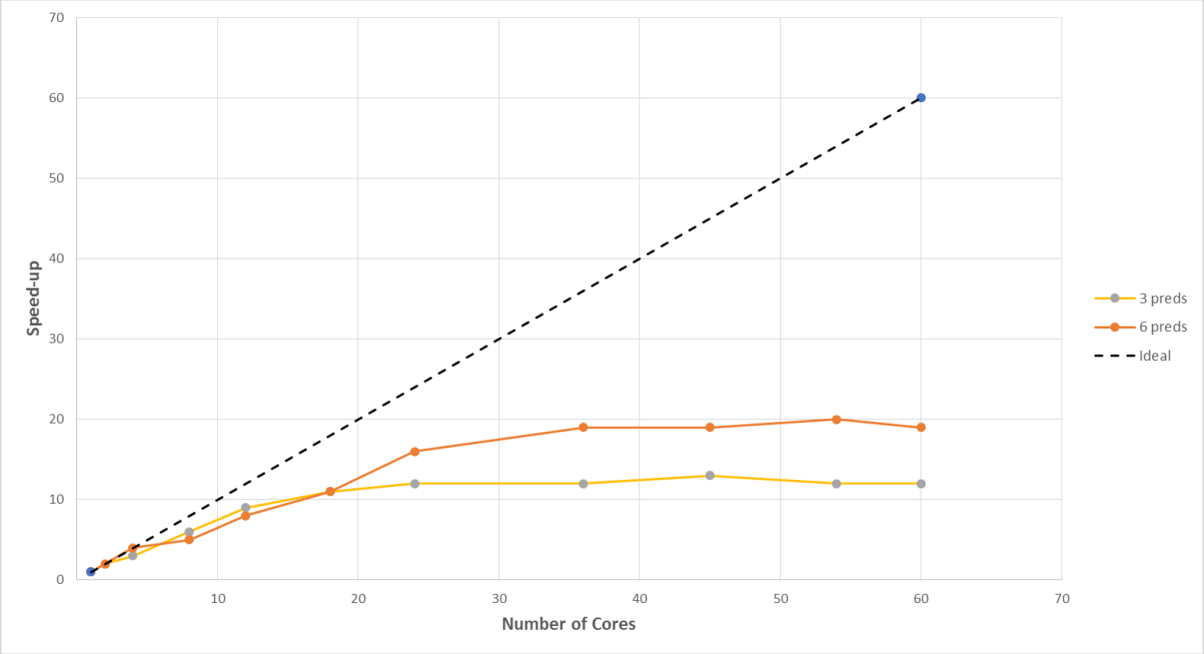


Figure 5.5: **Prey-capture**. The speed-up of the parallel Multi-Agent ESP implementation in the prey-capture task for a varying number of cores.

We observe a maximum speed-up of 20x at 54 cores for six predators. We once again observe

linear speed-up (with an R-squared value of 0.99) for the range of cores up to 24 cores. This indicates that the parallel processing framework is likely to work for more complex tasks (such as tasks with a large number of predators), on more cores. An expected decline in the speed-up to 19x is observed at 60 cores, due to scheduling and communication overheads in the goroutines [45]. When the maximum speed-up is achieved, for the six predator's version of the task, the runtime for the serial version is 20 minutes versus approximately one minute for the parallel version. This indicates that the task is harder with more agents (six predators). An interesting observation is that over 36 cores both task versions (three predators and six predators) converge at a runtime of approximately one minute. This indicates that the speed-up is attributed to the evaluations being spread across multiple cores and the number of predators has little effect on the runtime, past a certain number of cores. This speed-up confirms the viability of parallelization of the Multi-Agent ESP method, and demonstrates the complexity of the prey-capture task.

5.3 Summary

The maximum speed-up of 16x for the parallel ESP implementation confirms the feasibility of its parallelization. The application of the parallel processing framework to the multi-agent method, Multi-Agent ESP further confirms this. A maximum speed-up of 20x is achieved. The speed-ups achieved in the evaluation of the CCNE methods in the complex versions of the benchmark tasks demonstrates the parallel processing framework's ability to speed-up complex tasks.

Neuron SANE, our single population NE implementation, also achieves performance gains. However we observe the benefit of CC, in that ESP achieves a faster evolution time in the simpler double pole balancing task, and also converges at a maximum speed-up faster than neuron SANE. The results demonstrate the effectiveness of the parallel processing framework, in reducing evolution time, as well as its generality: single agent (for example ESP), as well as multi-agent (for example Multi-Agent ESP) methods are successfully parallelized.

The actual runtimes for the serial and parallel experiments detailed in this chapter are shown in Appendix D.

6 Conclusions

We found that the CCNE methods: ESP and Multi-Agent ESP, are well suited to multi-core acceleration with the ESP implementation yielding a maximum speed-up of 16x in the non-Markov version of the double pole balancing task and the Multi-Agent ESP implementation yielding a maximum speed-up of 20x in the prey-capture task, over their serial versions. We also found that non-CC methods, such as our implementation of neuron SANE can also benefit from multi-core acceleration, however a greater benefit would be to CCNE methods which can solve tasks more efficiently and can also solve more complex tasks, for example ones that require the memory capabilities of recurrent networks or ones that are highly dimensional (large search space). These results demonstrate the feasibility of the parallel processing framework to CC methods. This framework has a number of advantages, namely: 1) it is general, in that it is applicable to single-agent as well as multi-agent systems, but also applicable to non-coevolutionary NE methods, 2) it is fast, as demonstrated by the performance gains achieved by the parallel implementations, 3) it is easily implemented in a multi-core programming language, Go, that alleviates the programmer from conventional multi-threaded programming but also enables it to be highly portable, as demonstrated by its validation on a single parallel computer running Mac OS and the benchmarking on a HPC cluster running a Linux operating system, 4) it is also expressed in the MapReduce paradigm which is a design familiar to many software developers and enhances the efficiency of the implementations and 5) it is scalable, as demonstrated by the scalability of the Go implementations, that is, their ability to be applied to tasks of increasing complexity.

The serial and parallel implementations of SANE, ESP and Multi-Agent ESP have potential reusability as they are structured as packages (core packages and simulation environments), and can be used to implement other methods, and are not limited only to CC methods. The speed-up that may have been attained from the parallelisation of the recombination phase (evolution of the neuron populations) was purposely omitted from this study. This was done to focus primarily on the speed-up achieved from spreading the evaluations across multiple cores, which is the most compute intensive phase of the evolutionary process. This was confirmed by the effect of increasing the number of agents (predators) in the prey-capture task, we observed that the number of agents had little effect on the runtime of the task, particularly when more cores were utilized. Future enhancements could explore the parallelization of the evolution of the neuron population, particularly when the population size is large or there are a large number of agents. This work explored a number of theoretical foundations, such as varying population size, the effect of increasing the number of agents in the prey-capture task and the effect of varying the short pole length for the double pole balancing task, and what effect those had on task complexity. These are theoretical foundations that are important in this field of research [34]. Future work can explore other theoretical foundations such as convergence analysis (convergence to a benchmark task solution). Other interesting areas of future work may be the application of the CC framework to a real world real-time system [66, 9].

We observed linear speed-ups for the hard versions of the benchmark tasks over a certain range

of cores, this indicates to us that the Go implementations are efficient and that the harder the task the better the parallel speed-up. The performance gains demonstrated in this study provide a valuable foundation for the parallelization of even more complex CC methods within the NE domain, as well as within other CC domains.

References

[1] Enrique Alba, Gabriel Luque, and Sergio Nesmachnow. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[2] Tom Axford. *Concurrent programming: fundamental techniques for real time and parallel software design*. John Wiley & Sons, Inc. New York, NY, USA, 1989.

[3] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846, 1983.

[4] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, 2013.

[5] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[6] Chun-Chi J. Chen and Risto Miikkulainen. Creating Melodies with Evolving Recurrent Neural Networks. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, volume 3, pages 2241–2246. IEEE, 2001.

[7] Wenxiang Chen and Ke Tang. Impact of problem decomposition on Cooperative Coevolution. In *2013 IEEE Congress on Evolutionary Computation*, pages 733–740. IEEE, 2013.

[8] Alex Conradie, Risto Miikkulainen, and Christiaan Aldrich. Adaptive Control utilising Neural Swarming. Technical report, 2002.

[9] Wesley Cox, Tim French, Mark Reynolds, and Lyndon While. A Cooperative Coevolutionary Algorithm for Real-Time Underground Mine Scheduling. pages 410–418. Springer, Cham, 2018.

[10] Ciprian Craciun, Monica Nicoara, and Daniela Zaharie. Enhancing the scalability of metaheuristics by cooperative coevolution. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010.

[11] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms. *ACM Computing Surveys*, 45(3):1–33, 2013.

[12] Ermira Daka and Gordon Fraser. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[14] Shifei Ding, Hui Li, Chunyang Su, Junzhao Yu, and Fengxiang Jin. Evolutionary artificial neural networks: a review. *Artificial Intelligence Review*, 39(3):251–260, 2013.

[15] Thomas W. D'Silva. *Evolving Robot Arm Controllers using the NEAT method*. Masters thesis, The University of Texas at Austin, 2006.

[16] Ágoston E. Eiben and Günter Rudolph. Theory of evolutionary algorithms: A bird's eye view. *Theoretical Computer Science*, 229:3–9, 1999.

[17] Ágoston E. Eiben and Jim Smith. *Introduction to Evolutionary Computing*. Springer, Berlin, Germany, 2003.

[18] Alaa I. El-Nashar. To Parallelize or Not to Parallelize, Speed Up Issue. *International Journal of Distributed and Parallel systems*, 2(2):14–28, 2011.

[19] David B. Fogel. An Overview of Evolutionary Programming. *Evolutionary Algorithms*, pages 89–109, 1999.

[20] Shlomo Geva and Joaquin Sitte. A cartpole experiment benchmark for trainable controllers - IEEE Control Systems Magazine. *Control Systems, IEEE*, 13(5):40–51, 1993.

[21] Faustino J. Gomez. *Robust Non-linear Control through Neuroevolution*. PhD thesis, The University of Texas at Austin, 2003.

[22] Faustino J. Gomez and Risto Miikkulainen. Solving Non-Markovian Control Tasks with Neuroevolution. *International Joint Conference on Artificial Intelligence*, 16:1356–1361, 1999.

[23] Faustino J. Gomez and Risto Miikkulainen. Active Guidance for a Finless Rocket using Neuroevolution. *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 2084–2095, 2003.

[24] Ananth Grama, Vipin Kumar, George Karypis, and Anshul Gupta. *Introduction to parallel computing*. Addison-Wesley, Essex, England, 2003.

[25] Brian Greer, Henri Hakonen, Risto Lahdelma, and Risto Miikkulainen. Numerical optimization with neuroevolution. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 1, pages 396–401. IEEE, 2002.

[26] Eldon R. Hansen. *Numerical Optimization of Computer Models (Hans-Paul Schwefel)*. John Wiley & Sons, Inc., New York, NY, USA, 1981.

[27] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1994.

[28] Thomas Haynes and Sandip Sen. Evolving Behavioral Strategies in Predators and Prey. *Adaptation and Learning in Multi–Agent Systems*, pages 113–126, 1996.

[29] Dori Hershgal. Intel Threading Building Blocks. *Intel Technology Journal*, vol. 12(issue 01):27–38, 2008.

[30] Geoffrey E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, vol. 40(Issue 1-3):185–234, 1989.

[31] Charles A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[32] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Michigan, 1975.

[33] John H. Holland and Judith S. Reitman. Cognitive systems based on adaptive algorithms. In *Pattern directed inference systems*, pages 313–329. Academic Press, New York, 1978.

[34] Kashif Hussain, Mohd Najib Mohd Salleh, Shi Cheng, and Yuhui Shi. Metaheuristic research: a comprehensive survey. *Artificial Intelligence Review*, (January):1–43, 2018.

[35] Borhan Kazimipour, Xiaodong Li, and A. K. Qin. Initialization methods for large scale global optimization. In *2013 IEEE Congress on Evolutionary Computation*, pages 2750–2757. IEEE, 2013.

[36] Borhan Kazimipour, Xiaodong Li, and A. K. Qin. Effects of population initialization on differential evolution for large scale optimization. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2404–2411. IEEE, 2014.

[37] David B. Kirk and Wen Mei W. Hwu. *Programming massively parallel processors: A hands-on approach, second edition.* Elsevier, 2013.

[38] Go Programming Language. Available: https://golang.org, 2018.

[39] Sean Luke, Keith Sullivan, and Faisal Abidi. Large scale empirical analysis of cooperative coevolution. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation - GECCO '11*, pages 151–152, 2011.

[40] Sedigheh Mahdavi, Mohammad Ebrahim Shiri, and Shahryar Rahnamayan. Metaheuristics in large-scale global continues optimization: A survey. *Information Sciences*, 295(C):407–428, 2015.

[41] Risto Miikkulainen. Neuroevolution. In *Encyclopedia of Machine Learning.* Springer, New York, 2010.

[42] Risto Miikkulainen, Eliana Feasley, Leif Johnson, Igor Karpov, Padmini Rajagopalan, Aditya Rawal, and Wesley Tansey. Multiagent Learning through Neuroevolution. pages 24–46, 2012.

[43] Geoffrey Miller and Dave Cliff. Co-evolution of pursuit and evasion I: Biological and game-theoretic foundations. *From animals to animats*, 4(May):506–515, 1994.

[44] David E. Moriarty and Risto Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, pages 11–12, 1996.

[45] Sebastian Nanz, Scott West, Kaue Soares Da Silveira, and Bertrand Meyer. Benchmarking usability and performance of multicore languages. *International Symposium on Empirical Software Engineering and Measurement*, pages 183–192, 2013.

[46] Sune S. Nielsen, Bernabé Dorronsoro, Grégoire Danoy, and Pascal Bouvry. Novel efficient asynchronous cooperative co-evolutionary multi-objective algorithms. *2012 IEEE Congress on Evolutionary Computation, CEC 2012*, 2012.

[47] Peter S. Pacheco. Parallel Programming with MPI. *Performance Computing*, pages 1–43, 1997.

[48] Mitchell A. Potter and Kenneth A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary computation*, 8(1):1–29, 2000.

[49] Melissa Annette Redford, Chun Chi Chen, and Risto Miikkulainen. Modeling the Emergence of Syllable Systems. Technical report, 1998.

[50] Mark Roth, Micah J. Best, Craig Mustard, and Alexandra Fedorova. Deconstructing the overhead in parallel applications. *Proceedings - 2012 IEEE International Symposium on Workload Characterization, IISWC 2012*, 1:59–68, 2012.

[51] Min Shi. An empirical comparison of evolution and coevolution for designing artificial neural network game players. *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 379–386, 2008.

[52] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. California, USA, ebook edition, 2012.

[53] Kenneth O. Stanley. Efficient Reinforcement Learning through Evolving Neural Network Topologies. *Evolutionary Computation*, 1(2):3, 2002.

[54] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, vol. 9(Issue 6):653–668, 2005.

[55] Kenneth O. Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, vol. 21:63–100, 2004.

[56] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54, 2005.

[57] Chi Cuong Vu, Huu Hung Nguyen, and Lam Thu Bui. A parallel cooperative coevolution evolutionary algorithm. *Proceedings - 2011 3rd International Conference on Knowledge and Systems Engineering, KSE 2011*, pages 48–53, 2011.

[58] Bernard Widrow. The original adaptive neural net broom-balancer. *IEEE International Symposium on Circuits and Systems*, 2:351–357, 1987.

[59] Alexis P. Wieland. Evolving Neural Network Controllers for Unstable Systems. *IJCNN-91-Seattle International Joint Conference on Neural Network*, vol. 2:667–673, 1991.

[60] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC - First Experiences with Real-World Applications. pages 859–870. Springer, Berlin, Heidelberg, 2012.

[61] Xin-She Yang. Nature-Inspired Mateheuristic Algorithms: Success and New Challenges. *Journal of Computer Engineering and Information Technology*, 01(01):1–3, 2012.

[62] Xin-She Yang, Suash Deb, Thomas Hanne, and Xingshi He. Attraction and diffusion in nature-inspired optimization algorithms. *Neural Computing and Applications*, pages 1–8, 2015.

[63] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[64] Chern Han Yong and Risto Miikkulainen. Coevolution of Role-Based Cooperation in Multi-Agent Systems. 1:1–19, 2010.

[65] Ivan Zelinka. A survey on evolutionary algorithms dynamics and its complexity - Mutual relations, past, present and future. *Swarm and Evolutionary Computation*, 25:2–14, 2015.

[66] Andrea Zimmer, Arthur Schmidt, Avi Ostfeld, and Barbara Minsker. Evolutionary algorithm enhancement for model predictive control and real-time decision support. *Environmental Modelling & Software*, 69(C):330–341, 2015.

Appendices

A Hardware Specifications

Model	MacBook Pro (Retina, 15-inch, 2017)
OS	Mac OS Sierra
Processor	2.8 GHz Intel Core i7
Cores	8
Memory	16 GB 2133 MHz LPDR3

Table A.1: **The specifications of a single multi-core computer**. A Mac OS computer was used to validate the implementations.

Model	DELL C6145
OS	SUSE Linux Enterprise Server 11 SP4
Processor	2300 MHz AMD Opteron 6274
Cores	64
Memory	128 GB 1600 MHz

Table A.2: **The specifications of one High Performance Computing (HPC) worker node**. An HPC cluster was used to benchmark the single-agent (SANE, ESP) and multi-agent (Multi-Agent ESP) implementations.

B Visualizations

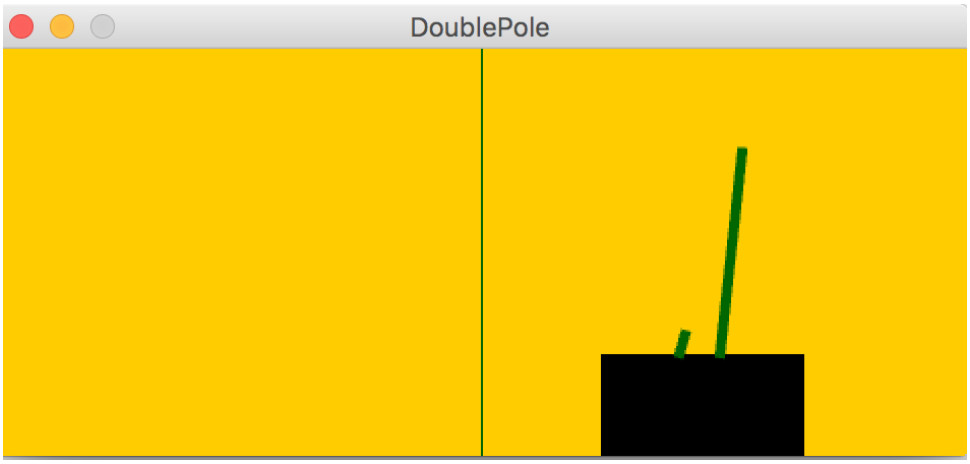


Figure B.1: **A snapshot of a controller evolved by the ESP method in the double pole balancing task**. The controller balances the two poles on the cart.

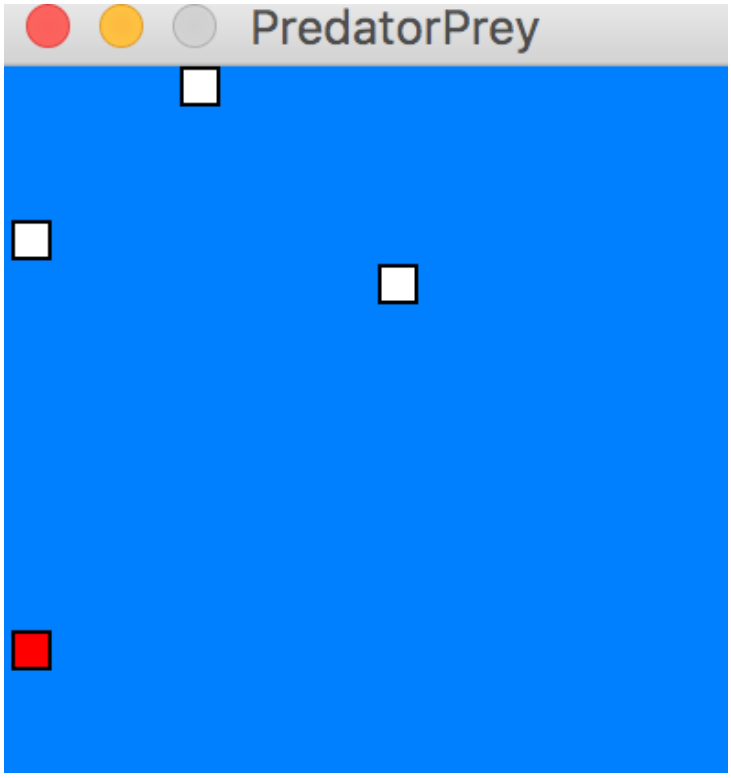


Figure B.2: **A snapshot of predators evolved by the Multi-Agent ESP method in the prey-capture task**. Predators are displayed in white and the prey in red.

C Runtimes for the validation experiments

	min (s)	max (s)	average (s)
SANE	2	10	6
ESP (Markov)	1	7	3
ESP (non-Markov)	5	21	13
Multi-Agent ESP	433	681	519

Table C.1: **The min, max and average timings in seconds for 10 validation experiments carried out on the HPC cluster for serial implementations of each method.** The parameters used for each of these experiments are detailed in Chapter 4.

D Runtimes for the benchmarking experiments

	0.6	0.7	0.8	0.85
1	1.5	1.4	1.4	1.3
2	0.7	0.7	0.7	0.5
4	0.5	0.4	0.4	0.4
8	0.3	0.3	0.3	0.3
12	0.3	0.3	0.3	0.3
15	0.3	0.3	0.3	0.3
20	0.3	0.3	0.3	0.3
24	0.3	0.3	0.3	0.3
30	0.3	0.3	0.3	0.3
40	0.3	0.3	0.3	0.3
50	0.3	0.3	0.3	0.3
60	0.3	0.3	0.3	0.3

Table D.1: **Actual runtime in seconds for the serial and parallel experiments for SANE.** The table shows the runtime for the serial version and for the parallel version, for each benchmarked number of cores and short pole length. The experiments were carried out on the HPC cluster.

	0.6	0.7	0.8	0.85
1	1.1	1	0.8	0.8
2	0.6	0.6	0.5	0.5
4	0.3	0.3	0.3	0.3
8	0.3	0.3	0.3	0.3
12	0.3	0.3	0.3	0.3
15	0.3	0.3	0.3	0.3
20	0.3	0.3	0.3	0.3
24	0.3	0.3	0.3	0.3
30	0.3	0.3	0.3	0.3
40	0.3	0.3	0.3	0.3
50	0.3	0.3	0.3	0.3
60	0.3	0.3	0.3	0.3

Table D.2: **Actual runtime in seconds for the serial and parallel experiments for ESP (Markov).** The table shows the runtime for the serial version and for the parallel version, for each benchmarked number of cores and short pole length. The experiments were carried out on the HPC cluster.

	0.6	0.7	0.8	0.85
1	9	18	30	47
2	6	8	14	27
4	3	4	5	17
8	3	3	5	8
12	2	3	4	7
15	2	3	3	5
20	2	2	3	3
24	2	2	3	4
30	2	3	4	6
40	2	3	4	5
50	2	4	4	5
60	2	3	4	4

Table D.3: **Actual runtime in seconds for the serial and parallel experiments for ESP (non-Markov).** The table shows the runtime for the serial version and for the parallel version, for each benchmarked number of cores and short pole length. The experiments were carried out on the HPC cluster.

	6 (predators)	3 (predators)
1	1194	751
2	673	417
4	319	254
8	224	126
12	153	84
18	104	66
24	75	64
36	63	63
45	63	57
54	60	62
60	64	64

Table D.4: **Actual runtime in seconds for the serial and parallel experiments for Multi-Agent ESP.** The table shows the runtime for the serial version and for the parallel version, for each benchmarked number of cores and number of predators. The experiments were carried out on the HPC cluster.

E Implementations

All Go implementations are publicly accessible:

`https://github.com/edmore/cooperative-coevolution`